Burroughs

Reference Manual

B 20 Operating System (BTOS)

(Relative to Release Level 3.0)

**Reference
Manual**

B 20 Operating System (BTOS)

*(Relative to Release Level 3.0)
Copyright © 1982, 1983 Burroughs Corporation, Detroit, Michigan 48232*

LIST OF EFFECTIVE PAGES

Page	Issue
iii thru xxv	Original
xxvi	Blank
1-1 thru 1-7	Original
1-8	Blank
2-1 thru 2-25	Original
2-26	Blank
3-1 thru 3-12	Original
4-1 thru 4-31	Original
4-32	Blank
5-1 thru 5-6	Original
6-1 thru 6-16	Original
7-1 thru 7-16	Original
8-1 thru 8-11	Original
8-12	Blank
9-1 thru 9-17	Original
9-18	Blank
10-1 thru 10-26	Original
11-1 thru 11-12	Original
12-1	Original
12-2	Blank
13-1 thru 13-7	Original
13-8	Blank
14-1 thru 14-82	Original
15-1 thru 15-40	Original
16-1 thru 16-10	Original
17-1 thru 17-34	Original
18-1 thru 18-12	Original
19-1 thru 19-17	Original
19-18	Blank
20-1 thru 20-4	Original
21-1 thru 21-28	Original
22-1 thru 22-6	Original
23-1 thru 23-18	Original
24-1 thru 24-16	Original
25-1 thru 25-10	Original
26-1 thru 26-28	Original
27-1 thru 27-3	Blank
27-4	Original
28-1 thru 28-20	Original
29-1 thru 29-21	Original
29-22	Blank
30-1 thru 30-6	Original
A-1 thru A-49	Original
A-50	Blank
B-1 thru B-10	Original
C-1 and C-2	Original
D-1 thru D-5	Original
D-6	Blank
E-1 thru E-12	Original
F-1 thru F-19	Original
F-20	Blank
G-1 thru G-41	Original
G-42	Blank
1 thru 20	Original

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

LIST OF ILLUSTRATIONS

LIST OF TABLES

INTRODUCTION

This manual provides descriptive and operational information for the B 20 Operating System (BTOS), hereinafter referred to as "Operating System" or "OS". The OS is a powerful, real time multitasking operating system for the B 20 Micro-Computer Systems. The information is provided in sections and appendices as listed in the Table of Contents. This information is relative to BTOS Release Level 3.0.

The following technical manuals are referenced for additional information:

Title

B 20 System Programmers and Assembler Reference Manual (Part 2)
B 20 Installation Planning Guide
B 20 Operations (Part 1)
B 20 Operations (Part 2)
B 20 Word Processing Quick Reference Guide
B 20 Pascal Reference Manual
B 20 FORTRAN Reference Manual
B 20 COBOL II Reference Manual
B 20 Systems Debugger Reference Manual
B 20 Systems Editor Reference Manual
B 20 Systems Linker/Librarian Reference Manual
B 20 System Programmers and Assembler Reference Manual (Part 1)
B 20 Systems Font Reference Manual
B 20 Systems Form Reference Manual
B 20 Systems ISAM Reference Manual
B 20 2780; 3780 RJE Reference Manual
B 20 3270 Reference Manual
B 20 Asynchronous Terminal Emulator (ATE) Reference Manual
B 20 Systems Sort/Merge Reference Manual
B 20 System Software Operation Guide

Form numbers and release level numbers for the above manuals can be found in the Customer Technical Publications Catalog and Price List - Form 1130010.

Software Patches

Within a particular release, patches to individual items may be issued. For example, an Operating System identified by 2.02.03 contains certain improvements over an Operating System 2.02.01. A patch always increases the patch number. All system software items within a given release (mark and level numbers) may be used together, regardless of the patch number, unless explicitly stated otherwise in the technical notes of the item.

The file [D0]<Sys>Sys.Version will be used to record the patches made to the software on the B20. It will be "Appended" if a new patch release is issued. The format of the file records will be as follows:

```
AAA BBBBB X.XX.XX
 |    |    |
 |    |    |_____ release level
 |    |_____ affected file
 |_____ Operating System or
                      utility identifier
```

For example, if a change to the Spooler was made, the record will look as follows:

        Spooler InstallSpl.Run 2.2.4-USA

CONVENTIONS USED IN THIS MANUAL

Numbers

Numbers are decimal except when suffixed with "h" for hexadecimal. Thus, 10h = 16 and 0FFh = 255.

Memory Address

Memory address refers to the logical memory address. (See the "Memory Management" section.)

Variable Names

Variables are named according to a formal convention. Some of the characteristics of the variable can be inferred from its name. Parameters used in procedure definitions and fields of request blocks and other data structures are named according to this convention.

A variable name is composed of up to three parts: a prefix, a root, and a suffix.

Prefixes

The prefix identifies the data type of the variable. Common prefixes are:

b   byte (8-bit character or unsigned number),

c   count (unsigned number),

f   flag (TRUE = 0FFh or FALSE = 0),

i   index (unsigned number),

n   number (unsigned number) (same as "c"),

o   offset from the segment base address (16 bits),

p   logical memory address (pointer) (32 bits consisting of the offset and the segment base address),

q   quad (32-bit unsigned integer),

rg   array of..., and

s   size in bytes (unsigned number).

Prefixes can be composed. Common compound prefixes are:

cb   count of bytes (the number of bytes in a string of bytes),

pb   pointer to (logical memory address of) a string of bytes, and

rgb   array of bytes.

Roots

The root of a variable name can be unique to that variable, selected from the list below, or a compound of the two. Common roots are:

dcb   Device Control Block,

dh   device handle,

erc   status (error) code,

exch   exchange,

fcb   File Control Block,

fh   file handle,

lfa   logical file address,

ph   partition handle

qeh   queue entry handle

rq   request block, and

ucb   User Control Block.

Suffixes

The suffix identifies the use of the variable. Suffixes are:

Last the largest allowable index of an array,

Max   the maximum length of an array or buffer (thus one greater than the largest allowable index), and

Ret   identifies a variable whose value is to be set by the called process or procedure rather than specified by the calling process.

Examples

Here are a few examples of variable names:

cbFileSpec   the count of bytes of a file specification,

ercRet   the status code to be returned to the calling process,

pbFileSpec   the memory address of a string of bytes containing a file specification,

pDataRet   the memory address of an area into which data is to be returned to the calling process,

ppDataRet   the memory address of a 4-byte memory area into which the memory address of a data item is to be returned to the calling process,

pRq   the memory address of a request block,

psDataRet   the memory address of a (2-byte) memory area into which the size of a data item is to be returned,

sData   the size (in bytes) of a data area,

sDataMax   the maximum size (in bytes) of a data area, and

ssDataRet   the size of the area into which the size of a data item is to be returned.

SECTION 1
OVERVIEW

MULTIPROGRAMMING

The B20 Operating System provides a real-time, multiprogramming environment. Multiprogramming is supported at three levels: application systems, tasks, and processes.

First, any number of application systems can coexist, each in its own memory partition. (An application system is a collection of one or more tasks that access a common set of files and implement a single application.)

Second, any number of tasks can be loaded into the memory of a partition and independently executed. (A task is an executable program, created by translating one or more source programs into object modules and linking them together.)

Third, any number of processes can independently execute the code (instructions) of each task. (A process is the basic element of computation that competes for access to the processor.)

EVENT-DRIVEN PRIORITY SCHEDULING

To meet the system builder's need for high performance, the Operating System Kernel provides efficient, event-driven, priority scheduling for an unlimited number of processes.

Each process is assigned one of 255 priorities and is scheduled for execution based on that priority. Whenever an event, such as the completion of an input/output operation, makes a higher priority process eligible for execution, rescheduling occurs immediately. This provides a more responsive system than scheduling techniques that are entirely time based.

To give multiple tasks with the same priority a fair share of system resources, processes with priorities in a predefined range are subject to time slicing. Processes with the same priority are then executed in turn for intervals of 100 ms in round robin fashion.

INTERPROCESS COMMUNICATION

The other major function provided by the OS Kernel is the interprocess communication (IPC) facility. IPC is used for synchronizing process execution and for transmitting information between processes.

A process can "send" a message and can "wait" for a message. When a process waits for a message, its execution is suspended until a message is sent to it. This allows processes to synchronize execution. A process can also "check" whether a message is available without its execution being suspended.

As a simple example, Process A sends a message to Process B and then waits for an answer. Process B waits for a message, performs a function determined by that message, and then sends an answering message. This sequence assures that Process B does not begin its function until requested and that Process A does not resume execution until Process B has completed its function.

As a more complex example, Process A continues execution in parallel with the execution of Process B before synchronizing execution by waiting for the answer.

EXCHANGES

Messages are not sent directly from process to process. Rather, they are routed through an intermediary element called an exchange.

Expanding on the example above: Process A sends a message to Exchange X and waits at Exchange Y, while Process B waits at Exchange X and sends an answering message to Exchange Y.

A single process can serve several exchanges, in which case it can select which of several kinds of messages to process next. This can be used to set priorities for the work the process is to perform.

Also, several processes can serve the same exchange, thereby sharing the processing of a single kind of message.

SYSTEM SERVICE PROCESSES

The B20 Operating System includes a number of system service processes. These processes, which are scheduled for execution in the same manner as application processes, receive IPC messages to request the performance of their services. Because of this internal use of IPC, the Operating System is classified as message-based.

Each system service process acts as the guardian and manager for a class of system resources such as files, memory, or keyboard. Because the system service process is the only software element that accesses the resource, and because the interface to the system service process is formalized through the use of IPC, a highly modular environment results.

This modular environment increases reliability by localizing the scope of processing and provides the flexibility to replace a system service process as a complete entity.

ACCESSING SYSTEM SERVICES

Each of the functions provided by the system service processes can be accessed through the use of a procedure call from high-level languages such as FORTRAN and Pascal, as well as from assembly language.

The use of a procedural interface masks all the complexities of using IPC: the procedural interface automatically uses a default response exchange and builds the "request block" message on the stack of the calling process.

In high-performance applications, however, the direct use of IPC operations to access system services allows an increased degree of concurrency between multiple input/output operations and computation.

FILTERS

Requests for system services are directed to the appropriate system service process through reference to a table that can be modified. This allows a system service request to be redirected to another system service process and also allows the implementation of filters. A filter enables the system builder to customize the function of a system service without modifying the system service process that implements it.

As an example, a filter process positioned between the file management system and its client processes can perform special password validation before permitting access to a file.

LOCAL RESOURCE-SHARING NETWORK (CLUSTER)

The Operating System provides support for local resource-sharing networks (clusters), as well as for standalone workstations. In a cluster configuration (consisting of a master workstation and up to 16 cluster workstations), essentially the same Operating System executes in each cluster workstation as in the master workstation. The master workstation provides file system and queue management resources for all workstations in the cluster. In addition, it concurrently supports its own interactive application processing.

In the cluster configuration, the IPC facility is extended to provide transparent access to system service processes that execute in the master workstation. While some services, like file management, queue management, 3270 emulator, and data base management, migrate to the master workstation, others, such as video and keyboard management, remain at the cluster workstation.

One high-speed RS-422 channel is standard on each workstation. This channel is used by cluster workstations for communication with the master workstation. Master workstations of small cluster configurations (up to four cluster workstations) use this channel for communications with their cluster workstations. However, master workstations of large cluster configurations use one or two Communications I/O Processors (CommIOPs) for communications with their cluster workstations.

The CommIOP, which is added to the Multibus of the master workstation, is an intelligent communications processor based on the Intel 8085 microprocessor. The CommIOP serves up to four cluster workstations on each of its two high-speed serial lines.

STANDARD NETWORK

A Standard Network extends the OS resource-sharing capabililiy to permit sharing of file system and printer spooler resources between clusters connected by leased, voice-grade lines and/or an X.25 Value-Added Network. In addition, the Standard Network permits access to other computers through the Value-Added Network.

VIRTUAL CODE SEGMENT MANAGEMENT

The OS virtual code segment management facility permits the execution of an application system whose size exceeds the available partition memory. To ensure maximum real-time performance, the use of this facility is under control of the system builder; an application system uses virtual code segment management only if the option is selected when its task image is linked.

If the virtual code segment facility is selected for a task, the code of the task is divided into variable-length segments that reside on disk. As the task executes, only the code segment being executed at a particular time must occupy the main memory of the partition. However, to maximize performance, recently used code segments are retained in memory as long as possible. Also, the data of the task remains in the main memory of the partition for the duration of task execution.

FILE MANAGEMENT

The OS file management system provides a hierarchical organization by volume, directory, and file. A volume is automatically recognized when placed online. Each file can have a 50-character file name, a 12-character password, and a file protection level. A file can be dynamically expanded or contracted without limit as long as it fits on one disk. Concurrent file access is controlled by read (shared) and modify (exclusive) access modes.

1-5

While providing convenience and security, the OS file management system supplies the system builder with the full throughput capability of the disk hardware. This includes reading or writing any sector of any open file with one disk access, reading or writing up to 64k bytes with one disk access, input/output overlapped with process execution, and optimized disk arm scheduling.

The duplication of critical <u>volume control structures</u> protects the integrity of disk file data against hardware malfunction. The Backup Volume utility is able to recover a file if either of its redundant <u>File Header Blocks</u> is valid.

DEVICE HANDLERS

A device handler can be part of the application process or it can be a system service process. Its interrupt handler can let the OS Kernel save process context (in which case it can be written in FORTRAN or Pascal), or it can receive the interrupt directly from the hardware. IPC provides an efficient, yet formal, interface from interrupt handler to device handler and from device handler to application process.

OTHER FEATURES

The Operating System also provides support for video display with multiple split screens, unencoded keyboard, communications lines, Sequential Access Method, Record Sequential Access Method, Direct Access Method, and Indexed Sequential Access Method.

COMMAND INTERPRETER

Interaction with the workstation operator is a function of the B20 Executive, not of the Operating System. This allows the system builder to choose the manner in which the video display and keyboard are used.

The Executive is a forms-oriented command interpreter providing an operator interface that includes a HELP facility, command files, and the interactive addition of new commands. The Executive is available for program development and for system builders that find its operator interface compatible with their users' needs. However, the Executive is a normal application-level program that can easily be replaced by the customized command interpreter of the system builder.

See the B20 System Executive Reference Manual, form 1144474 for more information about the Executive.

COMPACT SYSTEM

A compact version of the Operating System can be created at system build. The compact version requires less memory yet provides all Operating System functions except the simultaneous execution of multiple application systems. In the compact version, one application system is executed at a time.

BATCH MANAGER

Sequential execution of noninteractive application systems is a function of the batch manager. The batch manager interprets job control language files that execute specified application systems with specified parameters. The batch manager is useful for both program development and end-user environments.

SECTION 2
CONCEPTS

GENERAL

Some of the concepts described in this Section are illustrated in program examples in Appendix F.

STRUCTURE OF THE B 20 OPERATING SYSTEM

The basic components of the B20 Operating System are:

o   the Kernel,

o   system service processes,

o   system common procedures,

o   object module procedures, and

o   device and interrupt handlers.

The Kernel, the most primitive yet most powerful component of the Operating System, provides process management and interprocess communication facilities. It schedules process execution, including the saving and restoring of process context. A process is the basic element of computation that competes for access to the processor. The Kernel's interprocess communication primitives are the primary building blocks for synchronizing process execution and transmitting information between processes.

System service processes are OS processes that guard and manage system resources. System service processes are scheduled for execution in the same manner as application processes.

The four major categories of system services are:

o   task management,

o   file management,

o   device management, and

o   memory management.

There are two ways to access OS system services. The more convenient is by a procedure call from a high-level language. The more primitive allows an increased degree of concurrency between multiple input/output operations and computation.

System common procedures are OS procedures that perform some common system functions. An example of a system common procedure is Exit, which terminates the execution of an application system. System common procedures are executed in the same context and at the same priority as the invoking process. The Video Access Method is an example of system common procedures.

Object module procedures are procedures that are supplied as part of an object module file. They are not part of the OS System Image itself. Most application systems require only a subset, not a full set, of these procedures. The desired subset is linked into the application task. The Sequential Access Method is an example of object module procedures.

Device handlers and interrupt handlers of the Operating System are accessed indirectly through the convenient interfaces of the system service processes.

System builders can easily include their own system service processes, system common procedures, device handlers, and interrupt handlers in the OS System Image at system build. System build is the name for the sequence of actions necessary to construct a customized OS System Image. System build is described in the B20 System Programmers and Assembler Reference Manual (Part 1), form 1148699.

PROCESSING CONCEPTS

Under the Operating System, an application system (see Figure 2-1) is the collection of all logical software elements (tasks) currently in a partition. These tasks can be loosely or tightly coupled, but all perform related portions of the same application. These tasks execute asynchronously.

A task consists of executable code, data, and one or more processes. The code and data can be unique to the task or shared with other tasks. A task is created by translating one or more source programs into object modules and then linking them together. This results in a task image that is stored on disk in a run file.
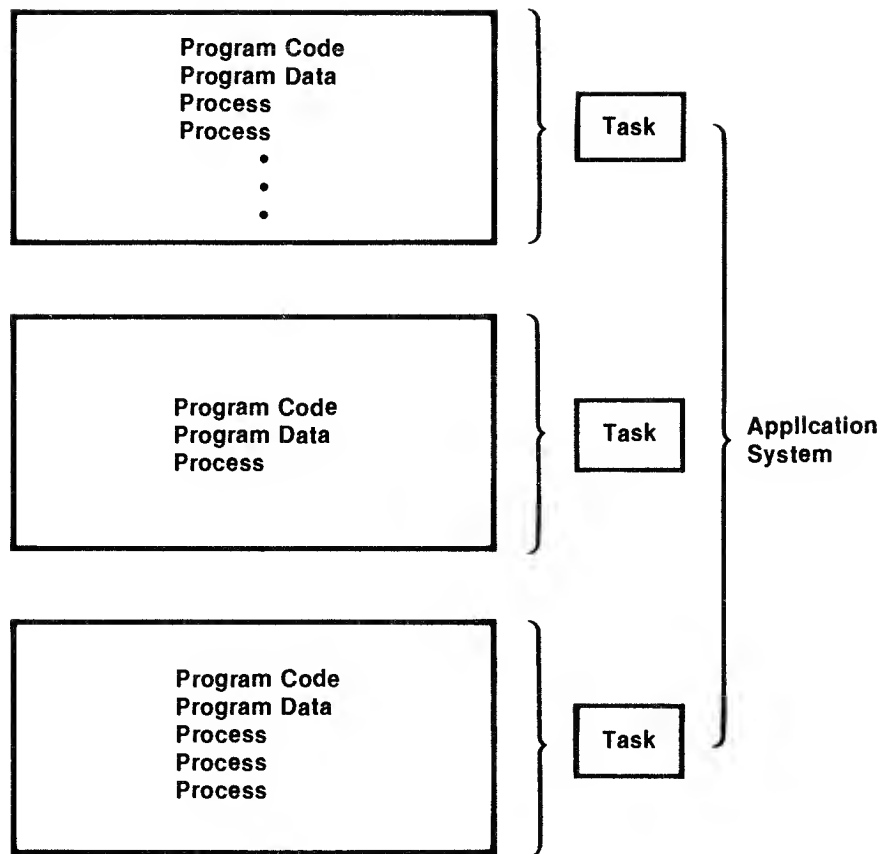
Figure 2-1. Relationship of Processes, Tasks, and an Application System

When requested by a currently active task, such as the Executive, the Operating System reads the task image from the run file into partition memory, relocates intersegment references, and schedules it for execution. The new task can coexist with or replace other application tasks in its partition memory.

A process is the basic element of computation that competes for access to the processor. A process consists of: (1) the address of the next instruction to execute on behalf of this process, (2) a copy of the data to be loaded into the processor registers before control is returned to this process, and (3) a stack. A process is assigned one of 255 priorities so that the Operating System can schedule its execution appropriately.

MEMORY ORGANIZATION

The memory of a system consists of two types of partitions:

system partitions, which contain the operating system and dynamically installed system services, and

application partitions, each of which contains an application system.

When a system is initiated, the Operating System is loaded into the system partition at the low address end of memory. Dynamically installed system services are loaded into extended system partitions located at the high address end of memory. The remaining memory is defined as a single application partition, called the primary application partition. (See Figure 2-2.)

Low End of Memory

System Partition

Interactive
Application
System

Primary Application Partition

Extended System Partition

High End of Memory
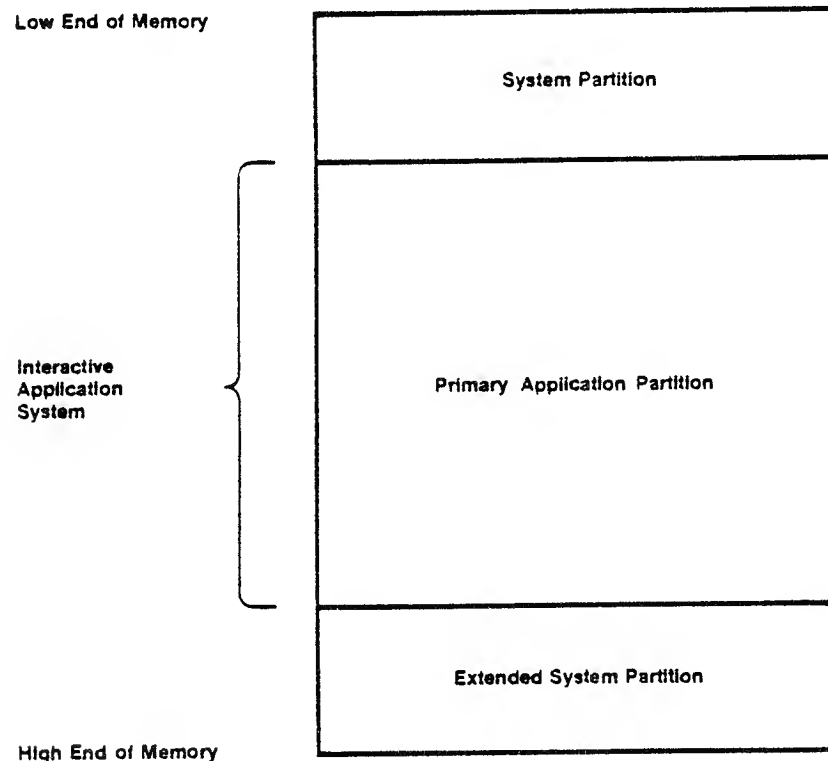
Figure 2-2. Memory Organization

When new partitions are created, they are placed at the high address end of the existing application partition and are called secondary application partitions. The remaining memory is defined as the primary application partition. (See Figure 2-3.)

The primary application partition is for interactive programs that use the keyboard and video display to interact with the user. Such partitions can be loaded with interactive programs chosen by the user, such as the Word Processor, a terminal emulator, or a user-written application program.

```
Low End of Memory          ┌─────────────────────────┐
                           │                         │
                           │    System Partition     │
                           │                         │
                           ├─────────────────────────┤
                           │                         │
Interactive                │                         │
Application                │  Primary Application    │
System                     │      Partition          │
                           │                         │
                           │                         │
                           ├─────────────────────────┤
                           │ Secondary Application   │
Noninteractive             │      Partition B        │
Application                ├─────────────────────────┤
Systems                    │ Secondary Application   │
                           │      Partition A        │
                           ├─────────────────────────┤
                           │                         │
                           │ Extended System Partition│
                           │                         │
High End of Memory         └─────────────────────────┘
```
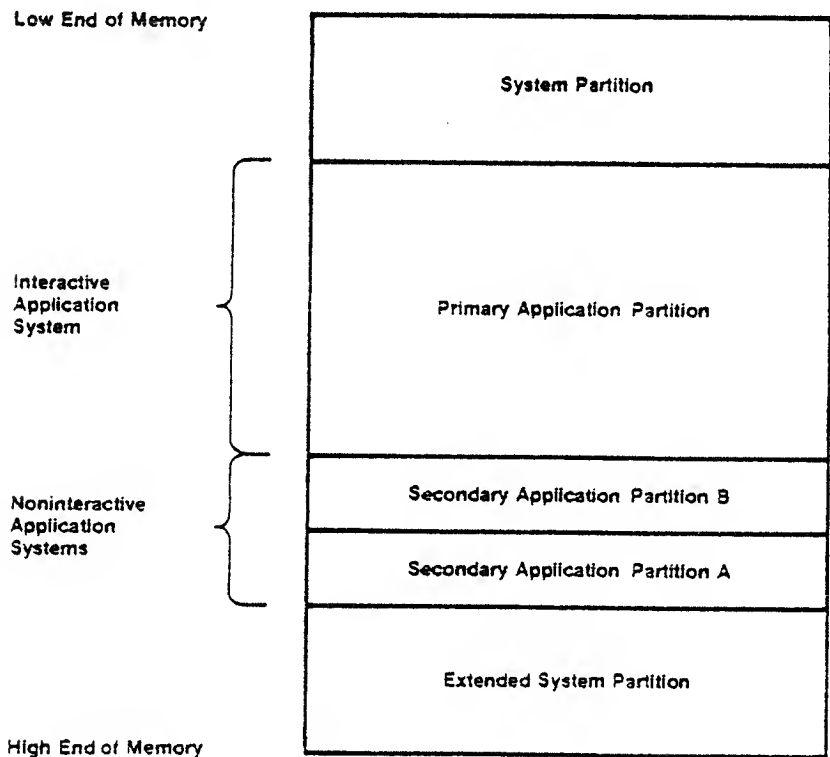
Figure 2-3. Memory Organization with Secondary Application Partition

2-5

Secondary application partitions are for noninteractive applications. Such partitions can be used for execution batch jobs under control of the batch manager, user-written applications, or system services.

A compact version of the Operating System can be built at system build that saves memory yet provides all Operating System functions described for the execution of one application system at a time. The compact version can have only one application partition, as shown in Figure 2-2.

Types of Memory

Two types of memory allocation are available to the application system: long-lived and short-lived. Within each application partition, long-lived memory expands upward from low memory locations while short-lived memory expands downward from high memory locations. The Operating System allocates short-lived memory for application tasks.

Processes within an application partition allocate and deallocate long-lived and short-lived memory by requests to OS system services. A process in one partition cannot allocate or deallocate memory in other partitions.

When the execution of an application system is terminated, the short-lived memory of its partition is automatically deallocated.

Long-lived memory is deallocated only at the explicit request of the application system. Therefore, long-lived memory is useful for passing information from an application system to a succeeding application system within the same partition.

VIRTUAL CODE SEGMENT MANAGEMENT

Virtual code segment management supports the execution of an application system whose size exceeds the available memory in its application partition. Program code (but not data) can reside on disk while a task is executing. Only the code segment whose instructions are being executed at a particular time need occupy the main memory of an application partition. The remaining code segments of the application system

are automatically read into partition memory as needed. When necessary, the oldest code segment in partition memory is overlaid to make enough partition memory available for a new code segment.

INTERPROCESS COMMUNICATION

As a message-based operating system, the OS uses its interprocess communication (IPC) facility internally for synchronization of process execution and information transmission. The OS Kernel provides IPC primitives to facilitate the consistent but flexible exchange of information between processes. Processes can communicate with each other within or between application partitions.

Six IPC primitives are provided: Check, PSend, Request, Respond, Send, and Wait. Both Operating System (that is, system service) and application system processes use these primitives.

Messages and Exchanges

Messages and exchanges are used in IPC.

A message conveys information and provides synchronization between processes. Although only a single 4-byte data item is literally communicated between processes, this data item is usually the memory address of a larger data structure. The larger data structure is called the message.

An exchange is the path over which messages are communicated from process to process (or from interrupt handler to process). An exchange consists of two first-in, first-out queues: one of processes waiting for a message, the other of messages for which no process has yet waited.

Processes or messages (but not both) can be queued at an exchange at any given instant. If a process waits at an exchange at which messages are queued, then the message that was enqueued first is dequeued and its address given to the process; the process then continues execution. Similarly, if a message is sent to an exchange at which processes are queued, then the process that was enqueued first is dequeued, given the address of the message, and placed into the ready state.

The relationship of exchanges, messages, and processes is shown in Figure 2-4.
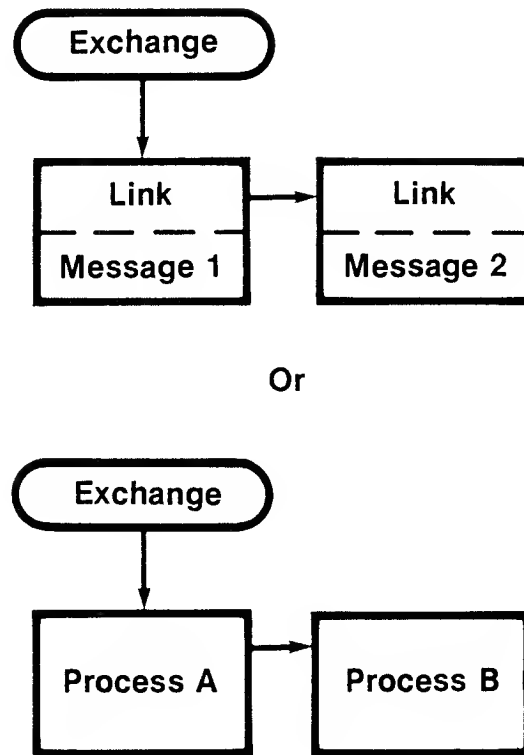
Figure 2-4. Relationship of Exchanges, Messages, and Processes

Exchanges are allocated in three ways:

o   at system build (for system service processes),

o   dynamically using the AllocExch (and DeallocExch) operation, and

o   at process creation.

A process can send a message to a process in another application partition. The destination process allocates an exchange and makes the exchange known to the OS. The sender process obtains the exchange number and sends messages to the exchange. Each of the processes must lock itself in its partition to prevent interference with the communication.

Process States

A process can exist in one of three states: running, ready, and waiting.

A process is in the running state when the processor is actually executing its instructions. Only one process can be in the running state at a time.

A process is in the ready state when it could be running, but a higher priority process is currently running. Any number of processes can be in the ready state at a time.

A process is in the waiting state when it is waiting at an exchange for a message. Any number of processes can be waiting at a time.

Table 2-1 describes the transitions between process states and the events causing the transitions. The relationship among process states is shown in Figure 2-5.

Table 2-1. Process State Transition

Transition

From	To	Event
Running	Waiting	A process executes a Wait but no messages are at the exchange.
Waiting	Ready/ Running	A process sends a message to the exchange at which a process is waiting.
Running	Ready	A higher priority process leaves the waiting state.
Ready	Running	All higher priority processes enter the waiting state.

```
        ┌──────────→        ┌────────────┐
        │     ┌─────────────────────┐     │
        │     │      Running        │     │
        │     │ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ │     │
        │     │      Ready          │     │
        │     └─────────────────────┘     │
        │              ↑        ↑         │
        │              │        └─────────┘
        │              │
        │     ┌─────────────────────┐
        │     │      Waiting         │
        │     └─────────────────────┘
        │              ↑
        └──────────────┘
```
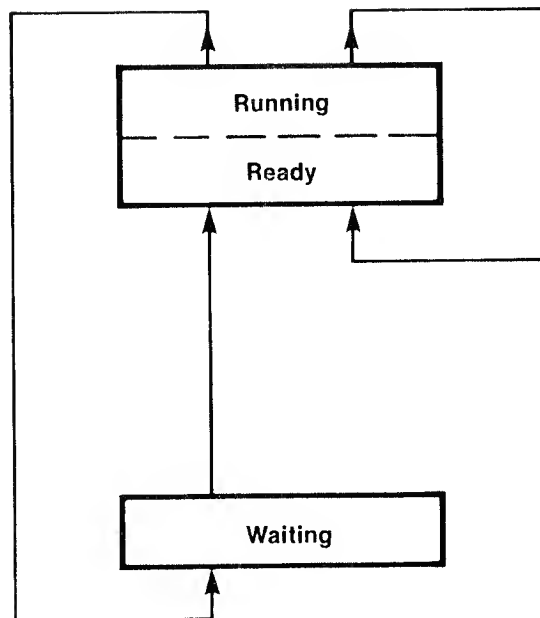
Figure 2-5. Process States

Process Priorities and Process Scheduling

Every process has a priority that indicates its importance relative to other processes. The priority of a process is assigned at process creation.

The Operating System has event-driven priority scheduling. This means that processes are scheduled for execution based on their priorities and system events, not on a time limit imposed by the scheduler. This involves very little decision-making of the OS. The scheduler maintains a queue of the processes that are eligible to execute. Priority determines which process, among those eligible, is executed. At any time, the OS always allocates the processor to the highest priority process that can be executed. Rescheduling occurs when a system event makes executable a process with a higher priority than the one currently executing.

A system event affects the executability of a process.   Examples of system events are an interrupt from a device controller, Multibus device, timer, or real-time clock, or a message sent from another process.   The system event causes a message to be sent to an exchange at which a higher priority process is waiting; this, in turn, causes the OS to reallocate the processor.

When a system event occurs that makes a process eligible to execute, the process receives control of the processor until another higher priority process preempts its execution, or until it voluntarily relinquishes control of the processor.

If no other process has work to perform, the null process, which executes at a priority (255) lower than any real process and which is always ready-to-run, is given control of the processor.   The null process exists only to simplify the algorithm of the scheduler; it performs no other useful work.

To give multiple tasks with the same priority a fair share of system resources, processes with priorities in a predefined range are subject to time slicing.   Such processes with the same priority are executed in turn for intervals of 100 ms in round robin fashion.

Sending a Message

When a message is sent to an exchange, the Operating System queues the address of the message, not the message itself.   Because only the address is moved, overhead is minimized, and queueing a number of messages at the same exchange requires little execution time or memory.

When a process sends a message to an exchange, one of two actions results at the exchange:

If no processes are waiting, the message is queued.

If one or more processes are waiting, the process that was enqueued first is given the message and is placed into the ready state. If this process has a higher priority than the sending process, it becomes the running process and the sending process loses control until it once again becomes the highest priority ready process.

After a message is queued at an exchange, it must not be modified by the sending process. A process that receives the message by waiting at the exchange where the message was queued is free to modify the message.

Waiting for a Message

When a process waits for a message at an exchange, one of two actions results at the exchange:

If no messages are queued, the process is placed into the waiting state until a message is sent. When a message is sent, its address is returned to the process, which leaves the waiting state and is scheduled for execution.

If one or more messages are queued, the message that was enqueued first is dequeued and its address returned to the process, which continues to execute.

Applying Interprocess Communication

To a large extent, the power of the Operating System results from its interprocess communication facility. IPC supports three multitasking capabilities:

o   communication,

o   synchronization, and

o   resource management.

Communication

Communication, the most elementary interaction between processes, is the transmission of data from one process to another via an exchange. Figure 2-6 below shows an example of communication between Process A and Process B. Process A sends a message to Exchange X, and Process B waits for a message at that Exchange.
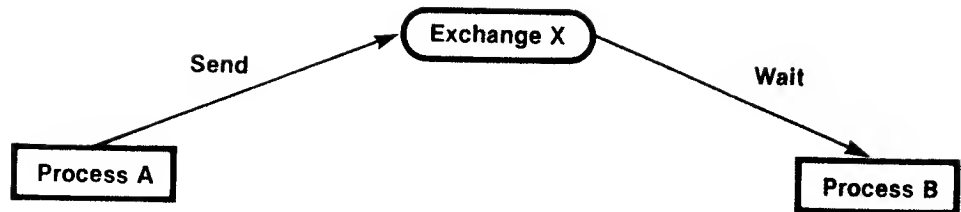
Figure 2-6. Communication between Processes

Synchronization

Synchronization is the means by which a process ensures that a second process has completed a particular item of work before the first process continues execution. Synchronization between processes and the transmission of data between processes usually occur simultaneously.

As shown in Figure 2-7 below, Process A sends a message to Exchange Y, requesting that Process B perform an item of work. Process A then waits at Exchange Z until Process B has completed the work. This synchronizes the continued execution of Process A with the completion of an item of work by Process B.
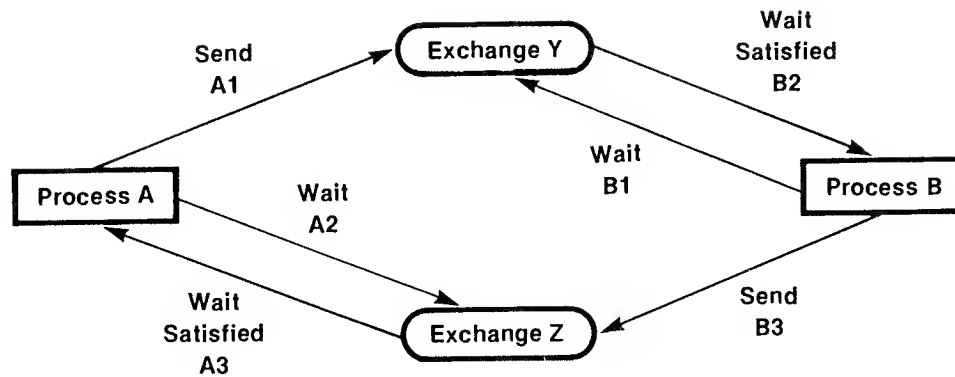
Figure 2-7. Synchronization

Resource Management

In a multitasking environment, <u>resource management</u> is the means of sharing resources among several processes in a controlled way. For example, several processes may need to use the printer; however, only one process can use the printer at a particular time.

One way to control a resource is to establish a process to manage it. Only the managing process accesses the resource directly. Other processes access the resource indirectly by sending messages to the process that performs the desired function. OS system services, which manage resources such as files, devices, and memory, are implemented via an analogous mechanism.

B 20 SYSTEM SERVICES

The Operating System includes a number of system
service processes. These processes, which are
scheduled for execution in the same manner as
application processes, receive IPC messages to
request the performance of their services. Any
process, even a system service process, can use
(be a client of) a system service process.

Each system service process acts as the guardian
and manager for a class of system resources such
as files, memory, or keyboard.

OS system services can be accessed:

indirectly, by a procedural interface, or

directly, by the Request and Wait primitives.

Using the procedural interface is easier because
it automatically performs most of the necessary
housekeeping and issues the Request and Wait
primitives.

Using the Request and Wait primitives is more
powerful, however, as it allows a greater degree
of overlap between multiple input/output
operations and computation.

Procedural Access to System Services

When a procedural interface is used, a request
block is automatically constructed and the
default response exchange of the process is
automatically used. (Request block and default
response exchange are defined immediately
below.) Except for the ReadAsync and WriteAsync
procedures, the request block is constructed on
the stack of the client process.

Direct Access to System Services

Execution of a system service involves the
participation of two processes (client and system
service), three kinds of Kernel primitives
(Request, Respond, and Wait), two kinds of
exchanges (response exchange and default response
exchange), and a data structure (request block).

2-15

The process requesting the system service is the
client process. Any process, even a system
service process, can be a client process, since
any process can request system services.

OS system services are provided by system service
processes. These processes are created when the
system is first loaded and execute code that was
linked into the System Image at system build.

A request block, a data structure provided by the
client process, contains the specification of,
and the parameters to, the desired system
service. A request block contains a request code
field, a response exchange field, and several
other fields.

A request code is a 16-bit value that uniquely
identifies the desired system service. For
example, the request code for the Write operation
is 36. The request code is used both to route a
request to the appropriate system service process
and to specify to that process which of the
several services it provides is currently
requested.

A response exchange is the exchange at which the
requesting client process waits for the response
of a system service. The response can be
directed to the exchange at which the client
process is expecting it because the exchange at
which the response is desired is specified in the
request block.

A special case of response exchange is the
default response exchange of a process. Each
process is given a unique default response
exchange when it is created. This special
exchange is automatically used as the response
exchange whenever a client process uses the
procedural interface to a system service.

A service exchange is an exchange that is
assigned to a system service process at system
build. The system service process waits for
requests for its service at its service exchange.

The Request primitive is a variant of the Send
primitive. It is used to direct a request for a
system service from a client process to the
service exchange of the system service process.
Request, unlike Send, does not accept an exchange

identification as a parameter. Rather, it infers the appropriate service exchange by using the request code as an index into the Service Exchange Table.

The Service Exchange Table is constructed at system build, resides in the System Image, and translates request codes to service exchanges.

The Respond primitive is another variant of the Send primitive. System service processes use Respond to report the completion of the requested system service.

Interaction of Client Processes and System Service Processes

The client process initiates the transaction by formatting a request block and issuing a Request primitive. After issuing the Request primitive, the client process can continue execution but must not modify the request block.

In order to determine when the request was completed, the client process must issue either a Wait or a Check primitive. The Wait or Check primitive must specify the same exchange that the client process specified as the response exchange in the request block.

The Wait primitive suspends execution of the client process until the system service process responds (or until another message is queued at the specified exchange).

The Check primitive does not suspend execution of the client process; instead it inquires whether a message is queued at the specified exchange.

The system service process waits for a request to be queued at an exchange. Upon receiving a request, the system service process verifies the control information and data given it before processing the request. After performing the requested function, it acknowledges completion of the service by responding to the client process. It then resumes waiting until it receives the next request.

The interaction of client and system service
processes is shown in Figure 2-8. The processing
flow of client and system service processes is
shown in Figure 2-9.
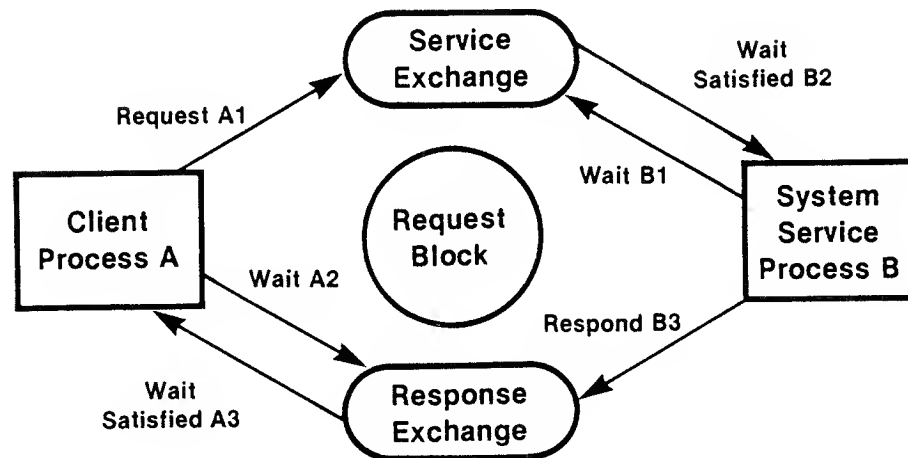
Figure 2-8. Interaction of Client and System Service Processes

```
Process Entry                          Process Entry
Point                                  Point


Initialize Process                     Initialize Process


Compute                                Wait for Request


Request Service                        Perform Function


Wait for Response                      Respond
```

CLIENT PROCESS   SERVICE PROCESS
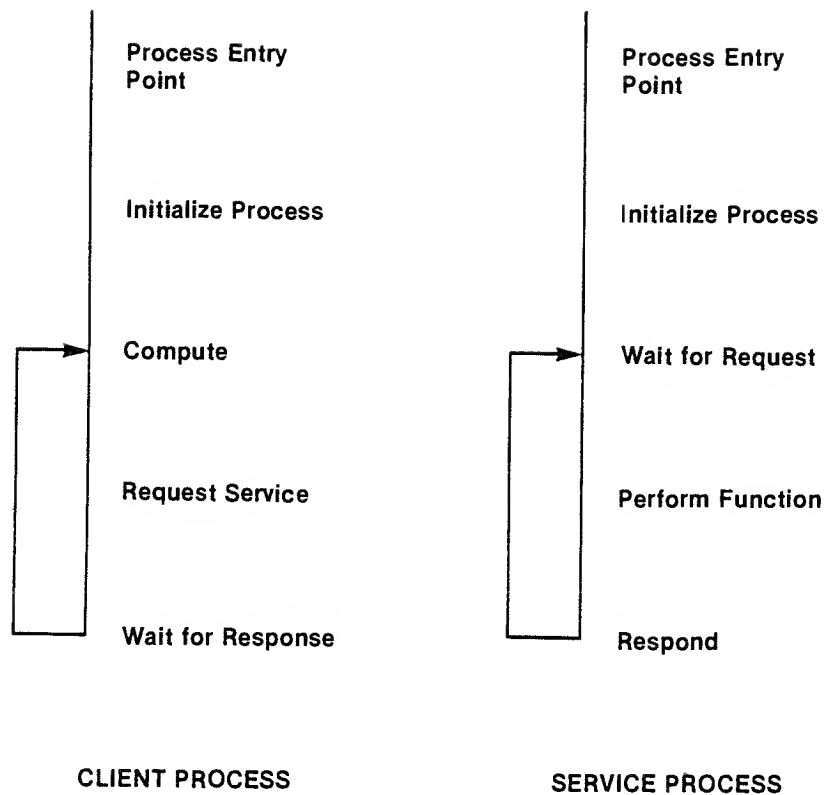
Figure 2-9. Processing Flow of Client and System Service Processes

Filter Processes

A filter process is a user-written system service process that is included in the System Image at system build. A filter process is interposed between a client process and a system service process that believe they are communicating directly with each other. The Service Exchange Table is adjusted at system build to route requests through the desired filter process.

A filter process might be used between the file management system and its client process to perform special password validation on all or some requests.

The interaction of a filter process with a client process and system service process is shown in Figure 2-10 below.
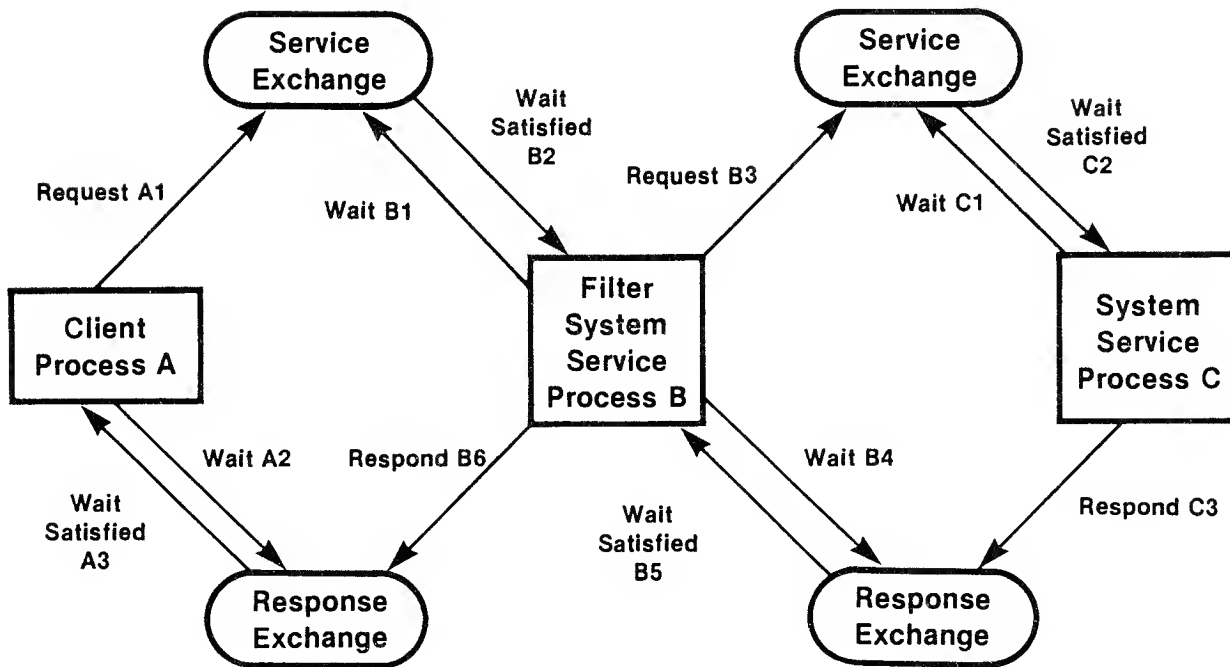
Figure 2-10. Interaction of Filter Process with Client and System Service Processes

Request Blocks

The format of request blocks is designed to allow the transparent migration of system service processes between standalone and cluster configurations. Request blocks are completely self-describing and consist of four parts:

1. a standard header,

2. request-specific control information,

3. descriptions of the request data items, and

4. descriptions of the response data items.

Each data item is described by memory address, size, and source (client or system service process).

CLUSTER CONFIGURATION

Cluster configurations of the B20 Series of Business Computer Systems consist of a master workstation and up to 16 cluster workstations. Essentially the same Operating System executes in each cluster workstation as in the master workstation. The master workstation provides file system and queue management resources for all workstations in the cluster. In addition, it concurrently supports its own interactive application processing as well as user-written multiuser system services. A cluster workstation can have its own local file system and printer spooler.

In the cluster configuration, the IPC facility is extended to provide transparent access to system service processes that execute in the master workstation. While some services, like file management, 3270 terminal emulator, and data base management, migrate to the master workstation, others, such as video and keyboard management, remain at the cluster workstation.

Application systems access the file system of a master workstation exactly as they do that of a standalone workstation. A program that works on a standalone workstation (accessing the local file system) can be moved to a cluster workstation (accessing the file system of the master workstation) without modification, recompilation, or relinking.

Interstation Communication

The interstation communication (ISC) facility is an upward-compatible extension of the interprocess communication facility. When a client process requests a system service, a request block is constructed that contains all the information necessary to describe the desired function.

In a standalone workstation, the request block is queued at the exchange of the system service process that actually performs the desired function.

Cluster Workstation Agent Service Process

In a cluster workstation, however, if the function is to be performed at the master workstation, then the request block is queued at the exchange of the Cluster Workstation Agent Service Process. The Cluster Workstation Agent Service Process converts interprocess requests to interstation messages for transmission to the master workstation. The Cluster Workstation Agent Service Process is included at system build in a System Image that is to be used on a cluster workstation.

Master Workstation Agent Service Process

The System Image used at the master workstation is built to include a corresponding service process. This process, the master workstation Agent Service Process, reconverts the interstation message to an interprocess request that it queues at the exchange of the master workstation system service process that actually performs the desired function. Note that the Service Exchange Table that translates the request code to a service exchange at the master workstation is necessarily different from the table at the cluster workstation.

When the system service process at the master workstation responds, the response is routed through the master workstation Agent Service Process, the high-speed data link, and the cluster workstation Agent Service Process before being queued at the response exchange in the cluster workstation that was specified in the request block.

The format of request blocks is designed to allow the cluster workstation and master workstation Agent Service Processes to convert between interprocess requests and interstation messages efficiently and with no external information. Because request blocks are completely self-describing, the Agent Service Processes can transfer requests and responses between master workstation and cluster workstations without any knowledge of what function is requested or how it is to be performed.

An interstation request message consists of:

o   a header,

o   control information,

o   the size and actual text of each request data item, and

o   the maximum allowed size of each response data item.

An interstation response message consists of:

o   a status code, and

o   the actual size and text of each response data item.

The cluster workstation Agent Service Process forms an interstation request message by copying the header and control information from the request block, moving the actual text of the request data items into the message, and including a specification of the maximum allowed sizes of the response data items.

After receiving the interstation response message, the cluster workstation Agent Service Process stores the status code into the request block and moves the text of the response data items into the memory areas specified for them by the request block. This transformation scheme ensures that no redundant or extraneous information is transmitted between master workstation and cluster workstations.

Communications I/O Processor

One high-speed RS-422 channel is standard on each workstation. This channel is used by cluster workstations for communications with the master workstation. Master workstations of small cluster configurations (up to four cluster workstations) see this channel for communications with their cluster workstations. Master workstations of large cluster configurations use one or two Communications I/O Processors (CommIOPs) for communications with their cluster workstations.

The CommIOP, which is added to the Multibus of the master workstation, is an intelligent communications processor based on the Intel 8085 microprocessor. The CommIOP serves up to four cluster workstations on each of its two high-speed serial lines.

CommIOP software consists of an 8085 bootstrap-ROM program, the main CommIOP program (which executes in 8085 RAM), and a CommIOP handler (written in 8086 code) which executes in system memory under OS control.

Software Organization

An OS System Image built for a cluster workstation differs from an OS System Image built for a standalone workstation in the (optional) exclusion of the file management system and the disk handler, and the inclusion of the cluster workstation Agent Service Process.

An OS System Image built for a master workstation differs from an OS System Image built for a standalone workstation only in its inclusion of the master workstation Agent Service Process. The master workstation is the file server for the entire cluster configuration. However, this does not necessitate the use of a different file management system from the one used in the standalone workstations. In fact, the file management system of the Operating System is actually a multiuser file system, even in a standalone workstation.

User-Written Software in a Cluster Configuration

Concurrency is the major issue concerning application systems executing on cluster workstations. Preferred programming practice dictates that the client process of a system service always examines the status code returned by the system service. However, while a program that opens a file without considering the possibility of receiving status code 220 ("File in use") executes successfully on a standalone workstation, such a program fails intermittently when executed on a cluster workstation at the same time that a program in another workstation is modifying the same file.

Whether user-written system services are good candidates for supporting multiple client processes depends both on the function they perform and the generality with which they are written. As an example, consider a user-written handler for a special Multibus device. If it used the standard format for request blocks, the device handler could be relocated to the master workstation. However, if it did not include concurrency checks, the device handler might become confused when it received requests from two or more workstations.

STANDARD NETWORK

(To be supplied)

SECTION 3
PROCESS MANAGEMENT

OVERVIEW

The process management facility provides event-driven priority scheduling and dynamic creation of multiprocess tasks.

Within each task of the application system and within the OS itself, the basic element of computation that competes for access to the processor is a process. Every process is assigned a priority. At all times, the OS process management facility allocates the processor to the highest priority process currently requesting it.

CONCEPTS

Process

A <u>process</u> is the basic element of computation that competes for access to the processor and which the OS schedules for execution.

A task has a single process associated with it when it is first loaded. That single process can create additional processes using the CreateProcess operation. The additional processes created typically share the same code but have separate stacks. The degree and means of data sharing are application-specific.

Processes and tasks usually have a hierarchical relationship. However, processes can execute code in multiple tasks. The usual relationship of a process to the tasks of an application system is shown in Figure 3-1 below.
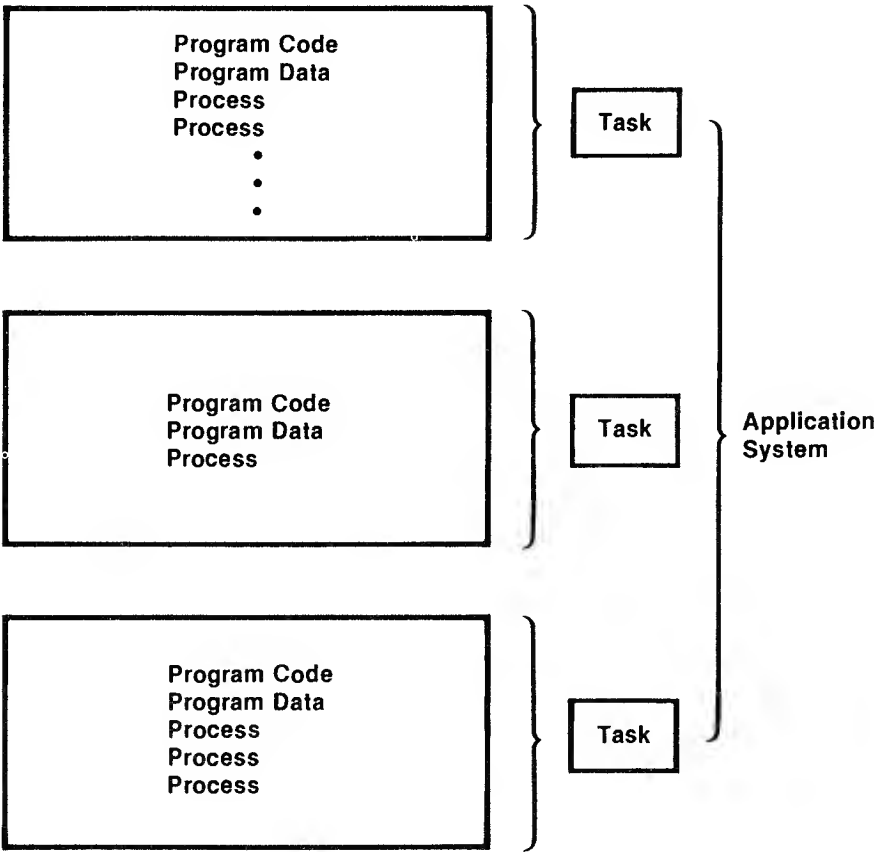
Figure 3-1. Relationship of Processes, Tasks, and an Application System

Context of a Process

The context of a process is the collection of all information about a process. The context has both hardware and software components.

The hardware context of a process consists of values to be loaded into processor registers when the process is scheduled for execution. This includes the registers that control the location of the process's stack.

The software context of a process consists of its default response exchange and the priority at which it is to be scheduled for execution.

The combined hardware and software context of a process is maintained in a system data structure called a Process Control Block (PCB). A PCB is the physical representation of a process.

When a higher priority process preempts a lower priority process, the OS saves the hardware context of the preempted process in that process's PCB. The OS later restores the contents of the registers when the process is rescheduled for execution; this permits the process to continue as though it were never interrupted. This is known as a context switch.

Process Priorities and Process Scheduling

Every process has a priority that indicates its importance relative to other processes. The priority of a process is assigned at process creation. Priorities range from 0 to 254 with 0 being the highest priority.

The OS has event-driven priority scheduling. This means that processes are scheduled for execution based on their priorities and system events, not on a time limit imposed by the scheduler. This involves very little decision-making for the OS. The scheduler maintains a queue of the processes that are eligible to execute. Priority determines which process among those eligible is executed. At any time, the OS always allocates the processor to the highest priority process that can be executed.

Rescheduling occurs when a system event makes executable a process with a higher priority than

the one currently executing. In most cases, the interval between events is determined by the duration of the typical input/output operation. A process never loses control involuntarily to another process of equal priority, only to a process of higher priority.

A system event affects the executability of a process. Examples of system events are an interrupt from a device controller, Multibus device, timer, or real-time clock, or a message sent from another process. The system event causes a message to be sent to an exchange at which a higher priority process is waiting; this, in turn, causes the OS to reallocate the processor.

When a system event occurs that makes a process eligible to execute, the process receives control of the processor until another higher priority process preempts its execution, or until it voluntarily relinquishes control of the processor.

If no other process has work to perform, the null process, which executes at a priority (255) lower than any real process and which is always ready-to-run, is given control of the processor. The null process exists only to simplify the algorithm of the OS scheduler; it performs no other useful work.

To give multiple tasks with the same priority a fair share of system resources, processes with priorities in a predefined range are subject to time slicing. Such processes with the same priority are executed in turn for intervals of 100 ms in round robin fashion. The priority range is a system build parameter, the default of which is 128 (80h) to 254 (FEh).

Process States

A process can exist in one of three states: running, ready, and waiting.

A process is in the running state when the processor is actually executing its instructions. Only one process can be in the running state at a time. Any other ready-to-run processes are in the ready state. As soon as the running process waits, the highest priority process in the ready state is placed into the running state and the execution context is switched to that process's context.

A process is in the <u>ready state</u> when it could be
running, but a higher priority process is
currently running. Any number of processes can
be in the ready state at a time.

A process is in the <u>waiting state</u> when it is
waiting at an exchange for a message. A process
enters the waiting state when it must synchronize
with other processes. A process can only enter
the waiting state by voluntarily issuing a Wait
primitive that specifies an exchange at which no
messages are currently queued. The process
remains in the waiting state until another
process (or interrupt handler) issues a Send (or
PSend, Request, or Respond) primitive that
specifies (indirectly in the case of
Request/Respond) the same exchange that was
specified by the Wait primitive. Any number of
processes can be waiting at a time. (See the
"Interprocess Communication Management" section
for more information on the Wait, Send, PSend,
Request, and Respond primitives.)

The relationship among process states is shown in
Figure 3-2 below.
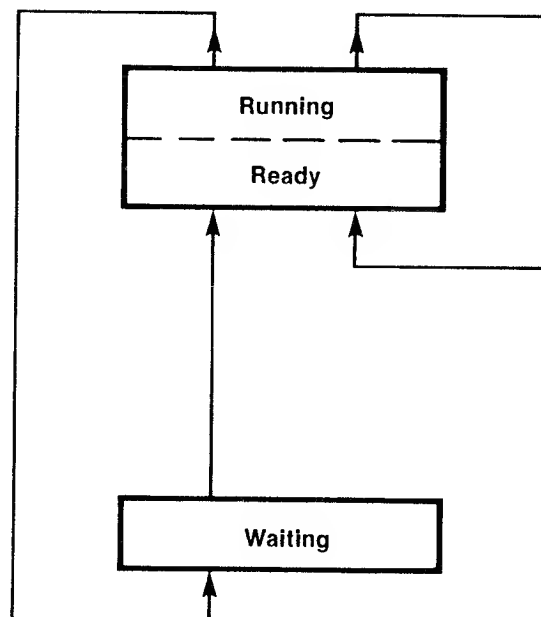
Figure 3-2. Process States

Table 3-1 below describes the transitions between process states and the events causing the transitions.

Table 3-1. Process State Transition

Transition From	To	Event
Running	Waiting	A process executes a Wait but no messages are at the exchange.
Waiting	Ready/ Running	Another process sends a message to the exchange at which a process is waiting.
Running	Ready	A higher priority process leaves the waiting state.
Ready	Running	All higher priority processes enter the waiting state.

OPERATIONS: PRIMITIVES AND PROCEDURES

Process management provides the operations listed below.

ChangePriority	changes the priority of the calling process.
CreateProcess	creates a new process and schedules it for execution.
GetUserNumber	allows a process to determine its own user number.
QueryProcessNumber	allows a process to determine its own process number.

ChangePriority

Description

The ChangePriority primitive changes the priority
of the calling process.

Procedural Interface

ChangePriority (priority): ErcType

where

priority   is the new priority.

Request Block

ChangePriority is a Kernel primitive.

CreateProcess

Description

The CreateProcess primitive creates a new process and schedules it for execution. CreateProcess is called by an application process to establish an application system in which multiple processes execute the same reentrant task code.

The CreateProcess primitive is also used by the Chain and LoadTask operations to create the initial process of a new task. (See the "Task Management" section.)

Procedural Interface

CreateProcess (pProcessDescriptor): ErcType

where

pProcessDescriptor
is the memory address of a Process Descriptor Block. The format for a Process Descriptor Block is shown in Table 3-2 below.

Request Block

CreateProcess is a Kernel primitive.

GetUserNumber

Description

> The GetUserNumber procedure allows a process to determine its own user number.

Procedural Interface

> GetUserNumber (pUserNumberRet):   ErcType
> where
> pUserNumberRet
> > is the memory address of a word into which the user number of the inquiring process is returned.

Request Block

> GetUserNumber is a system common procedure.

Table 3-2. Processor Descriptor Block

Offset	Size (bytes)	Field	Description
0	4	pEntry	Memory address (CS:IP) at which to begin execution of the new process.
4	2	saData	Segment base address to be loaded into the Data Segment (DS) register when the new process is scheduled for execution.
6	2	saExtra	Segment base address to be loaded into the Extra Segment (ES) register when the new process is scheduled for execution.
8	2	saStack	Segment base address to be loaded into the Stack Segment (SS) register when the new process is scheduled for execution.
10	2	oStackInit	Offset value to be loaded into the Stack Pointer (SP) register when the new process is scheduled for execution.
12	1	priority	Priority (0-254, with 0 the highest) at which the new process is to be scheduled for execution.

Table 3-2. Process Descriptor Block (Cont.)

Offset	Size (bytes)	Field	Description
13	1	fSys	Always FALSE. A value of TRUE would indicate that the new process was a system process and would cause a subsequent Chain operation to fail.
14	2	defaultResponseExchange	Identification of an exchange that the calling process has allocated using the AllocExch operation. (See the "Exchange Mangement" section.) This exchange becomes the default response exchange of the new process. The calling process must never use this exchange again in order to avoid possible conflict.
16	1	fDebug	Indicates whether the new process is to be debugged. TRUE indicates it will be debugged, and, therefore, is not to be scheduled for execution; FALSE indicates it is to be scheduled for execution. (See the B 20 Systems Debugger Reference Manual, form 1148665.)

QueryProcessNumber

Description

The QueryProcessNumber procedure allows a process to determine its own process number.

Procedural Interface

QueryProcessNumber (pProcessNumberRet): ErcType

where

pProcessNumberRet
> is the memory address of a word into which the process number of the inquiring process is returned.

Request Block

QueryProcessNumber is a system common procedure.

SECTION 4
INTERPROCESS COMMUNICATION MANAGEMENT

OVERVIEW

The interprocess communication (IPC) facility synchronizes process execution and information transmission between processes through the use of messages and exchanges. A process can communicate with another process in its own partition or in another application partition.

Messages

A process can send a message and wait for a message. When a process waits for a message, its execution is suspended until a message is sent to it. This allows processes to synchronize execution. A process can also check whether a message is available without its execution being suspended.

In its simplest form, IPC provides unidirectional transmission of arbitrary data. After preparing a data structure (a message) that is to be passed to another process, Process A uses the IPC facility to send the address of the message to Process B. Only the address of the message, not the message itself, is buffered by IPC. The size and content of the message are not constrained by IPC. Process B must be programmed to use the IPC facility to wait or check for the availability of a message.

The full power of IPC is best appreciated when pairs of unidirectional transmissions are matched.

As a simple example, Process A sends a message to Process B and then waits for an answer. Process B waits for a message, performs a function determined by that message, and then sends an answering message. This sequence assures that Process B does not begin its function until requested and that Process A does not resume execution until Process B has completed its function.

Since Process B does not send an answer until after it has processed the message, the answer can signal Process A that the message is no longer being used by Process B and (possibly)

that Process B has modified the message in a manner agreed upon by the two processes.

As a more complex example, Process A continues execution in parallel with the execution of Process B before synchronizing execution by waiting for the answer.

Exchanges

A message is sent to a system entity called an exchange rather than directly to a process. An exchange should be thought of as serving the function of a post office where postal patrons (processes) go to mail (send) letters (messages) or pick up (wait/check for) letters (messages).

In the same way that a postal patron drops a letter in the mailbox and then walks away trusting that the letter will be delivered, a process sends a message and then continues executing without further regard for the message.

A postal patron who is expecting an important letter can periodically go to the post office to check whether it has arrived. If the letter is especially important, the patron can wait in the post office for the letter to arrive.

A process has analogous mechanisms available when it expects to receive a message. It can periodically check whether a message is posted at (enqueued on) an exchange or it can wait at the exchange for the arrival of a message. Because computers are many orders of magnitude faster than the postal service, it is usually more appropriate to wait for a message than to check for its arrival.

A process can send a message to a process in another application partition. The destination process allocates an exchange, then makes the exchange known to the Operating System. The sender process obtains the exchange number and sends messages to the exchange. Each process must lock itself into its partition so it cannot be terminated.

System Service Processes

The Operating System includes a number of system service processes. These processes, which are scheduled for execution in the same manner as

application processes, receive IPC messages to request the performance of their services. Any process, even a system service process, can use (be a client of) a system service process.

Each system service process acts as the guardian and manager for a class of system resources such as files, memory, or keyboard. Because the system service process is the only software element that accesses the resource, and because the interface to the system service process is formalized through the use of IPC, a highly modular environment results.

This modular environment increases reliability by localizing the scope of processing and provides the flexibility to replace a system service process as a complete entity.

Accessing System Services

OS system services can be accessed:

o   indirectly, by a procedural interface, or

o   directly, by the Request and Wait primitives.

Using the procedural interface is easier because it automatically performs most of the necessary housekeeping, as well as issuing the Request and Wait primitives.

Using the Request and Wait primitives is more powerful, however, as it allows a greater degree of overlap between multiple input/output operations and computation.

When the processes of an application system use the Send and Wait primitives to communicate among themselves, they are free to structure their messages in whatever way is most convenient. They are also free to pair unidirectional transmissions into bidirectional transmissions using whatever conventions are convenient, or to use the IPC facility in a manner that does not involve pairing.

When communicating with OS system service processes, however, the rules are different. The concept of pairing two unidirectional transmissions into a bidirectional transmission is formalized and enforced. Also, the format of the message that is communicated is formalized.

The format of the message (a request block) is
designed to allow the transparent migration of
system service processes between standalone and
cluster configurations. Request blocks are
completely self-describing and consist of (1) a
standard header, (2) request-specific control
information, and (3) descriptions of the request
and response data items. Each data item is
described by memory address, size, and source
(client or system service process).

The Send primitive is not used to communicate
with OS system services. Rather, two other
primitives, Request and Respond, initiate the
request for a system service and its response.
This provides:

o   assurance that Requests and Responds are
    matched,

o   assurance that system resources are always
    available to transmit responses,

o   opportunity to redirect requests for system
    services to other system service processes,
    and

o   opportunity to redirect requests for system
    services to the master workstation of a
    cluster configuration.

Filter Processes

Requests for system services are directed to the
appropriate system service process through
reference to a table that can be modified. This
allows a system service request to be redirected
to another system service process and also allows
the implementation of filters. A filter enables
the system builder to customize the function of a
system service without modifying or even looking
at the system service process that implements it.

As an example, a filter process positioned
between the file management system and its client
process can perform special password validation
before permitting access to a file.

Cluster Configuration

In the cluster configuration, the IPC facility
is extended to provide transparent access to
system service processes that execute in the

master workstation. In the master workstation, the Operating System concurrently supports local application processing and resource sharing (disk and printer) for the other workstations of the cluster. While some services, like file management, queue management, 3270 terminal emulator, and data base management, migrate to the master workstation, others, such as video and keyboard management, remain at the cluster workstation.

CONCEPTS

The <u>interprocess communication</u> (IPC) facility provides process synchronization and information transmission through the use of messages and exchanges.

Messages

A <u>message</u> conveys information and provides synchronization between processes. Although only a single 4-byte data item is literally communicated between processes, this data item is usually the memory address of a larger data structure. The larger data structure is called the message while the 4-byte data item is conventionally called the <u>address of the message</u>. The message can be in any part of memory that is under the control of the sending process. By convention, control of the memory that contains the message is passed along with the message.

Exchanges

An <u>exchange</u> is the path over which messages are communicated from process to process (or from interrupt handler to process). An exchange consists of two first-in, first-out queues: one of <u>processes</u> waiting for a message, the other of <u>messages</u> for which no process has yet waited. An exchange is referred to by a unique 16-bit integer.

Processes <u>or</u> messages (but not both) can be queued at an exchange at any given instant. If a process waits at an exchange at which messages are queued, then the message that was enqueued first is dequeued and its memory address given to the process; the process then continues execution. Similarly, if a message is sent to an exchange at which processes are queued, then the process that was enqueued first is dequeued, given the address of the message, and placed into the ready state.

Link Blocks

Small system data structures (<u>link blocks</u>) are used for enqueueing messages onto an exchange.

Each link block contains the address of the
message and the address of the next link block
(if any) that is linked onto the exchange.
Processes are enqueued onto an exchange by
linking through a field of each Process Control
Block that is reserved for this purpose.

The relationship of exchanges, messages, and
processes is shown in Figure 4-1 below.

```
        ┌─────────────┐
        │  Exchange   │
        └──────┬──────┘
               │
               ▼
     ┌─────────────┐      ┌─────────────┐
     │    Link     │─────▶│    Link     │
     │ ─ ─ ─ ─ ─ ─ │      │ ─ ─ ─ ─ ─ ─ │
     │  Message 1  │      │  Message 2  │
     └─────────────┘      └─────────────┘
```

Or

```
        ┌─────────────┐
        │  Exchange   │
        └──────┬──────┘
               │
               ▼
     ┌─────────────┐      ┌─────────────┐
     │             │─────▶│             │
     │  Process A  │      │  Process B  │
     │             │      │             │
     └─────────────┘      └─────────────┘
```
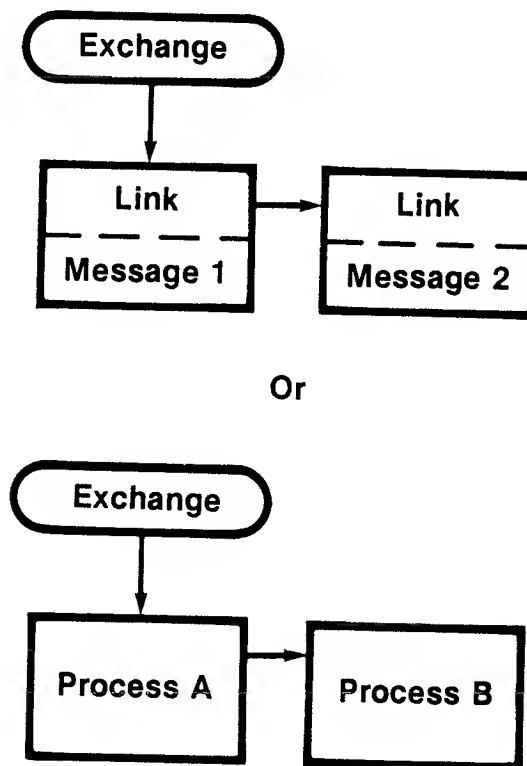
Figure 4-1. Relationship of Exchanges, Messages, and Processes

Exchanges are allocated in three ways:

o   Exchanges for system service processes are allocated at system build.

o   Exchanges can be dynamically allocated and deallocated using the AllocExch and DeallocExch operations. (See the "Exchange Management" section.)

o   When a process is created, its creator gives it a unique default response exchange. A process can determine the identification of its own default response exchange using the QueryDefaultRespExch operation. (See the "Exchange Management" section.)

Sending a Message

When a message is sent to an exchange, the OS queues the address of the message at the exchange. Thus overhead is minimized as just the address of the message, not the message itself, is moved. Therefore queueing a number of messages at the same exchange requires very little execution time or memory.

When a process sends a message to an exchange, one of two actions results at the exchange:

If no processes are waiting, the message is queued.

If one or more processes are waiting, the process that was enqueued first is given the message and is placed in the ready state. If this process has a higher priority than the sending process, it becomes the running process and the sending process loses control until it once again becomes the ready process with the highest priority.

After a message is queued at an exchange, it must not be modified by the sending process. A process that receives the message by waiting at the exchange where the message was queued is free to modify the message.

The Send primitive transfers a 4-byte field from the sending process to the waiting process. The 4-byte field can be interpreted as the memory address of a data structure but this is not necessary. The interpretation of the 4-byte field is by agreement of the two processes involved.

Waiting for a Message

When a process waits for a message at an exchange, one of two actions results at the exchange:

If no messages are queued, the process is placed in the waiting state until a message is sent. When a message is sent, its memory address is returned to the process, which leaves the waiting state and is scheduled for execution.

If one or more messages are queued, then the message that was enqueued first is dequeued and its memory address returned to the process, which continues to execute.

Sending Messages to Another Partition

A process can send a message to a process in another application partition (interpartition communication). The destination process first allocates an exhange with the AllocExch operation, then uses the SetPartitionExchange operation to make the exchange known to the OS. The sender process uses the GetPartitionExchange operation to obtain the exchange number, then sends messages to the exchange.

Each process must use the LockPartition operation to lock itself into its partition so that it cannot be terminated by a TerminatePartitionTasks or VacatePartition operation.

The AllocExchange operation is described in the "Exchange Management" section. The GetPartition-Exchange, LockPartition, SetPartitionExchange, TerminatePartitionTasks, and VacatePartition operations are described in the "Application Partition Management" section.

System Service Processes

The Operating System includes a number of system service processes. These processes, which are scheduled for execution in the same manner as application processes, receive IPC messages to request the performance of their services. Any process, even a system service process, can use (be a client of) a system service process.

Each system service process acts as the guardian and manager for a class of system resources such as files, memory, or keyboard. Because the system service process is the only software element that accesses the resource, and because the interface to the system service process is formalized through the use of IPC, a highly modular environment results.

This modular environment increases reliability by localizing the scope of processing and provides the flexibility to replace a system service process as a complete entity.

Accessing System Services

OS system services can be accessed:

o   indirectly, by a procedural interface, or

o   directly, by the Request and Wait primitives.

Using the procedural interface is easier because it automatically performs most of the necessary housekeeping, as well as issuing the Request and Wait primitives.

Using the Request and Wait primitives is more powerful, however, as it allows a greater degree of overlap between multiple input/output operations and computation.

Procedural Access to System Services

When a procedural interface is used, a request block is automatically constructed and the default response exchange of the process is automatically used. Except for the ReadAsync and WriteAsync procedures, the request block is constructed on the stack of the client process.

Most procedural interfaces to system services do not provide any overlap between computation by

the client process and execution of the system service. Because Read and Write are the system services for which the overlap of computation and execution of the system service is most desirable, the procedures ReadAsync and CheckReadAsync and WriteAsync and CheckWriteAsync have been provided. (See the "File," "Disk," and "Printer Spooler Management" sections.) These procedures allow the client process to initiate an input/output operation and then compute and/or initiate other input/output operations before checking for the successful completion of the input/output operation.

Direct Access to System Services

Execution of a system service involves the participation of two processes (client and system service), three kinds of Kernel primitives (Request, Respond, and Wait), two kinds of exchanges (response exchange and default response exchange), and a data structure (request block).

The process requesting the system service is the client process. Any process, even a system service process, can be a client process, since any process can request system services.

OS system services are provided by system service processes. These processes are created when the system is first loaded and execute code that was linked into the System Image at system build.

System services are customized at system build through the inclusion/exclusion of Burroughs written system service processes in the System Image. User-written system service processes can also be included, either to replace or to augment the Burroughs-written ones. User-written system service processes have the same power and flexibility as Burroughs-written ones; customizing the set of system services requires no modification of Burroughs-written code.

A request block, a data structure provided by the client process, contains the specification of and the parameters to the desired system service. A request block contains a request code field, a response exchange field, and several other fields that are explained in the section below on "Request Blocks."

A _request code_ is a 16-bit value that uniquely identifies the desired system service. For example, the request code for the Write operation is 36. The request code is used both to route a request to the appropriate system service process and to specify to that process which of the several services it provides is currently requested.

A _response exchange_ is the exchange at which the requesting client process waits for the response of a system service. The response can be directed to the exchange at which the client process is expecting it because the exchange at which the response is desired is specified in the request block.

A special case of response exchange is the _default response exchange_ of a process. Each process is given a unique default response exchange when it is created. This special exchange is automatically used as the response exchange whenever a client process uses the procedural interface to a system service.

For this reason, the direct use of the default response exchange is _not_ recommended. The use of the default response exchange is limited to requests of a synchronous nature. That is, the client process, after specifying the exchange in a Request, must wait for a response before specifying it again (indirectly or directly) in another Request.

A _service exchange_ is an exchange that is assigned to a system service process at system build. The system service process waits for requests for its services at its service exchange.

The _Request_ primitive is a variant of the Send primitive. It is used to direct a request for a system service from a client process to the service exchange of the system service process. Request, unlike Send, does not accept an exchange identification as a parameter. Rather, it infers the appropriate service exchange by using the request code as an index into the Service Exchange Table.

The _Service Exchange Table_ is constructed at system build, resides in the System Image, and translates request codes to service exchanges. A companion table, the _Local Service Code Table_,

translates each request code to a local service code to specify which of the several services of the system service process is desired.

The Respond primitive is another variant of the Send primitive. System service processes use Respond to report the completion of the requested system service. The exchange to which the response is directed is not a direct parameter to Respond but is obtained from the response exchange field of the request block. Only system service processes are allowed to use the Respond primitive, and they must always specify as a parameter the same request block that the client process used to request the system service.

Interaction of Client Processes and System Service Processes

The client process initiates the transaction by formatting a request block and issuing a Request primitive. The client process can then continue execution but must not modify the request block. In order to determine when the request was completed, the client process must issue either a Wait or a Check primitive. The Wait or Check primitive must specify the same exchange that the client process specified as the response exchange in the request block.

The Wait primitive suspends execution of the client process until the system service process responds (or until another message is queued at the specified exchange).

The Check primitive does not suspend execution of the client process; instead it inquires whether a message is queued at the specified exchange.

The system service process waits for a request to be queued at an exchange. Upon receiving a request, the system service process verifies the control information and data given it before processing the request.

If the request is invalid, the system service process inserts an appropriate error code into the status code field (that is, ercRet) of the request block.

If the request is valid, the system service process performs the request, places appropriate information into the response packets described by the request block, inserts a normal status code into the request block, and acknowledges

completion of the service by responding to the
exchange specified by the client process. It
then resumes waiting until it receives the next
request.

The interaction of client and system service
processes is shown in Figure 4-2 below.
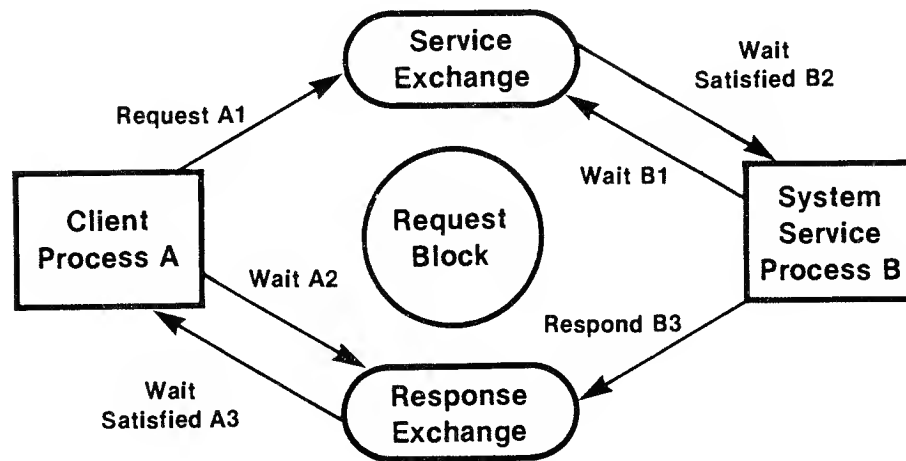
Figure 4-2. Interaction of Client and System Service Processes

The processing flow of client and system service
processes is shown in Figure 4-3.

Process Entry
Point

Initialize Process

Compute

Request Service

Wait for Response

CLIENT PROCESS

Process Entry
Point

Initialize Process

Wait for Request

Perform Function

Respond
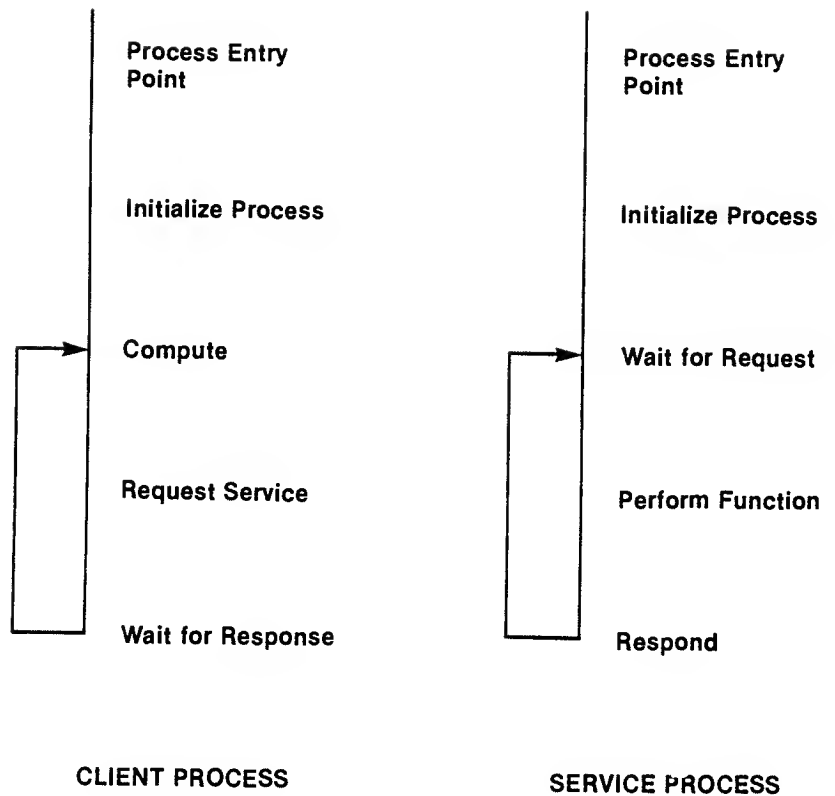
SERVICE PROCESS

Figure 4-3. Processing Flow of Client and System Service Processes

Filter Processes

A filter process is a user-written system service process that is included in the System Image at system build. A filter process is interposed between a client process and a system service process that believe they are communicating directly with each other. The Service Exchange Table is adjusted at system build to route requests through the desired filter process.

A filter process might be used between the file management system and its client process to perform special password validation on all or some requests.

The interaction of a filter process with a client and system service process is shown in Figure 4-4 below.
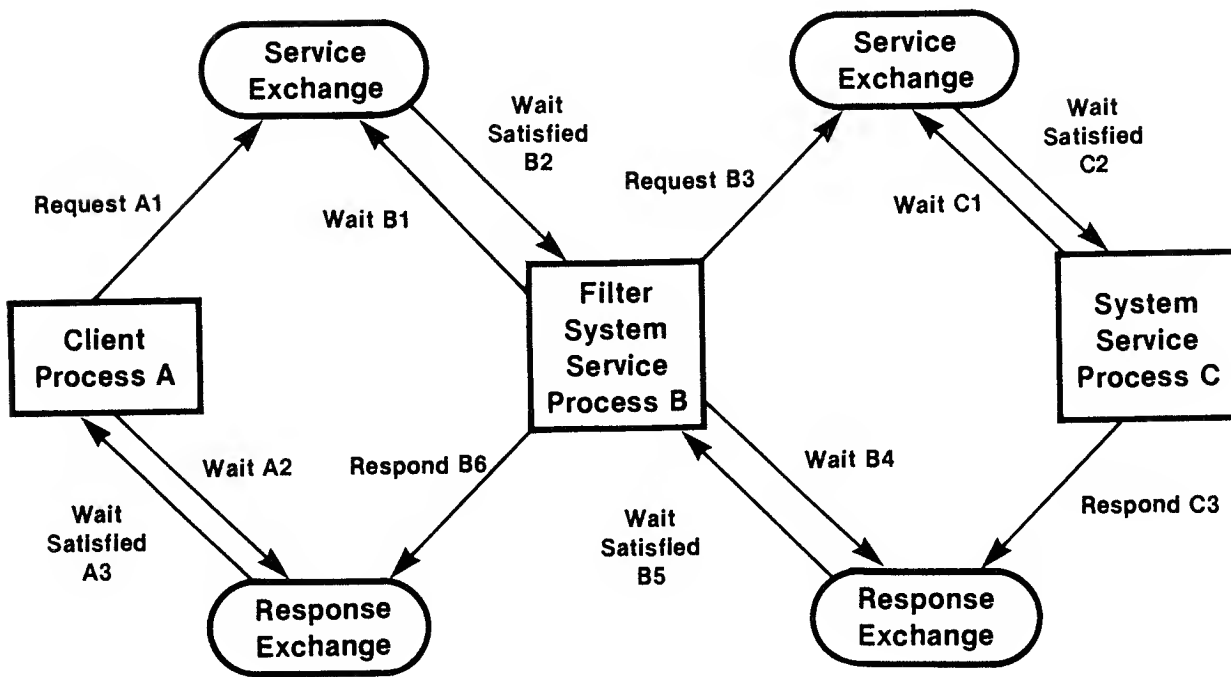
Figure 4-4. Interaction of Filter Process with Client and System Service Processes

Request Blocks

The format of request blocks is designed to allow the transparent migration of system service processes between standalone and cluster configurations. Request blocks are completely self-describing and consist of four parts:

1. a standard header,

2. request-specific control information,

3. descriptions of the request data items, and

4. descriptions of the response data items.

Each data item is described by memory address, size, and source (client or system service process).

Standard Header

The format of the standard request block header is shown in Table 4-1 below.

Table 4-1. Format of a Request Block Header

Offset	Field	Size (bytes)
0	sCntInfo	2
2	nReqPbCb	1
3	nRespPbCb	1
4	userNum	2
6	exchResp	2
8	ercRet	2
10	rqCode	2

---

where

sCntInfo   is the number of bytes of control information.

nReqPbCb   is the number of request address/size (pb/cb) pairs.

nRespPbCb   is the number of response address/maximum size (pb/cbMax) pairs.

userNum   is a 16-bit user number that uniquely identifies the application system. Each application partition has a unique user number. The processes in an application partition share the same user number. A process can obtain its user number with the GetUserNumber operation (see the "Process Management" section).

exchResp   is the response exchange. CAUTION: Be extremely careful in specifying the response exchange in the request block. Conflicting use

4-17

of exchanges, especially explicit use of the default response exchange of a process that conflicts with the implicit use by procedural calls to system services and system common procedures, tends to cause application systems to malfunction in ways that are difficult to diagnose.

ercRet   is the status code (returned by the system service process).

rqCode   is a request code, a 16-bit value that uniquely identifies the desired system service. The request code is used both to route a request to the appropriate system service process and to specify to that process which of the several services it provides is currently requested.

Request-Specific Control Information

The request-specific control information consists of sCntInfo bytes that are transmitted from client to system service (except for ercRet, which is transmitted from system service to client).

Request Data Item

Each request data item descriptor consists of the 4-byte memory address of the request data item followed by the 2-byte size of the item. The total size (in bytes) of the request data item descriptors is six times nReqPbCb. Request data items are transmitted from client to system service.

Response Data Item

Each response data item descriptor consists of (1) the 4-byte memory address of the area into which the response data item is to be moved by the system service, and (2) the 2-byte maximum allowable byte count of the response data item. The total size (in bytes) of the response data item descriptors is six times nRespPbCbMax. Response data items are transmitted from system service to client.

Example

As an example, consider a request to write one sector into a file that has already been opened. Assume that the client process is using a procedural interface rather than directly using the Request and Wait primitives. The client process makes a function reference (that is, erc = Write (fh, pBuffer, sBuffer, lfa, psDataRet);) to the file management system Write operation, supplying as arguments:

o   the file handle returned from a previous OpenFile operation,

o   the memory address of the first byte of data to be written,

o   the count of bytes to be written,

o   the logical file address of the sector into which the data is to be written, and

o   the memory address of the word into which the count of bytes successfully written is to be returned.

The Write function also returns a status code indicating the success of the operation.

The Write system service illustrates both a request data item (the data to be written) for which the client process is the source and a response data item (the count of bytes successfully written) for which the system service process is the source.

In this example, the procedural interface automatically allocates memory on the stack of the client process for a request block and creates a header containing:

o   the number of bytes of control information (6),

o   the number of items of request data (1),

o   the number of items of response data (1),

o   the user number (the default is 0 for the application system in the interactive partition),

o   the response exchange (the default response exchange of the client process is used

4-19

automatically whenever a system service is activated through its procedural interface),

o   the status code (this is returned by the system service process), and

o   the request code (36 is the request code to invoke the Write system service).

The control information contains:

o   the file handle (2 bytes), and

o   the logical file address (4 bytes).

The single request data item is described by:

o   the memory address of the data to be written, and

o   the count of bytes to be written.

The single response data item is described by:

o   the memory address of the word into which the count of bytes successfully written is to be returned, and

o   the size (in bytes) of the word into which the count of bytes successfully written is to be returned (the number 2 is automatically supplied by the procedural interface).

Request Primitive

The Request primitive is a variant of the Send primitive. It directs a request for a system service from a client process to the service exchange of the system service process.

The Send primitive accepts any 4-byte field as a parameter. This is usually, but not necessarily, the address of a message. In contrast, the Request and Respond primitives explicitly interpret the 4-byte field as the memory address of a request block. Before issuing the Request primitive, the client process arranges the data required for the system service into a request block in its memory.

Unlike Send, Request does not accept an identification of an exchange as a parameter. Rather, it infers the appropriate service

exchange by using the request code of the request block as an index into the Service Exchange Table. The Service Exchange Table is constructed at system build, resides in the System Image, and translates request codes to service exchanges.

The use of the Service Exchange Table allows request codes to remain invariant among Operating Systems with varying organizations of system service processes. This invariance facilitates the development of filters and is critical to the transparent operation of the cluster configuration.

A companion table, the Local Service Code Table, translates each request code to a local service code to specify which of the several services of the system service process is desired.

Respond Primitive

The Respond primitive is only used by a system service process to respond to a client process that requested the performance of a system service.

The only parameter to the Respond primitive is the memory address of the request block of the client process. That is, the system service must use the same memory address as a parameter to Respond that the client process used as a parameter to the Request primitive. The exchange to which the response is directed is determined by the response exchange (exchResp) field of the request block.

In normal operation, sufficient resources (that is, link blocks) are always available for the successful execution of the Respond primitive. This is because the Request primitive reserves a link block for the exclusive use of the corresponding Respond primitive. Calls to the Respond primitive must exactly match calls to the Request primitive. That is, each Request must be answered by a Respond, and Respond must never be used for any purpose other than to answer a Request.

If a malfunctioning user-written system service were to fail to respond to a client process's request, unmatched requests would cause all link blocks to be reserved and future requests would receive the "No link block available" status code.

If an application process inappropriately called the Respond primitive, the unmatched Respond would cause the count of link blocks reserved to be insufficient and might cause another call to Respond to receive the "No link block available" status code.

Wait Primitive

The Wait primitive is used with the Request and Respond primitives, as well as with the Send primitive. System service processes use Wait to suspend execution until a client process requests the performance of a system service. Client processes use Wait to synchronize their execution with the completion of the system service they requested. In the context of Request and Respond, the message that is queued at an exchange is always a request block.

The Wait primitive first checks whether one or more messages are queued at the specified exchange.

If messages are queued, then the message that was enqueued first is dequeued from the exchange and its memory address returned to the calling process; the calling process then continues execution.

If no messages are queued, the Process Control Block of the calling process is queued at the exchange and the process is placed into the waiting state. In the waiting state, the process stops executing and relinquishes control of the processor. The calling process remains in the waiting state until another process queues a message at the specified exchange. The calling process then leaves the waiting state and is placed into the ready state. The memory address of the message queued at the exchange by the other process is returned to the calling process and it resumes execution when it becomes the highest priority ready process.

Interstation Communication

The interstation communication (ISC) facility is an upward-compatible extension of the interprocess communication facility. When a client process requests a system service, a request block is constructed that contains all

the information necessary to describe the
desired function.

In a standalone workstation, the request block is
queued at the exchange of the system service
process that actually performs the desired
function.

Cluster Workstation Agent Service Process

In a cluster workstation, however, if the
function is to be performed at the master
workstation, the request block is queued at the
exchange of the cluster workstation Agent
Service Process. The cluster workstation Agent
Service Process converts interprocess requests
to interstation messages for transmission to the
master workstation. The cluster workstation
Agent Service Process is included at system
build in a System Image that is to be used on a
cluster workstation.

Master Workstation Agent Service Process

The System Image used at the master workstation
is built to include a corresponding service
process: the master workstation Agent Service
Process. The master workstation Agent Service
Process reconverts the interstation message to
an interprocess request that it queues at the
exchange of the master workstation system
service process that actually performs the
desired function. Note that the Service
Exchange Table that translates request code to
service exchange at the master workstation is
necessarily different from the table at the
cluster workstation. When the system service
process at the master workstation responds, the
response is routed through the master
workstation Agent Service Process, the
high-speed data link, and the cluster
workstation Agent Service Process before being
queued at the response exchange in the cluster
workstation that was specified in the request
block.

The format of request blocks is designed to
allow the cluster workstation and master
workstation Agent Service Processes to convert
between interprocess requests and interstation
messages very efficiently and with no external
information. Because request blocks are
completely self-describing, the Agent Service

4-23

Processes can transfer requests and responses between the master workstation and cluster workstation without any knowledge of what function is requested or how it is to be performed.

Interstation Request/Response Message

An interstation request message consists of:

o   a header,

o   control information,

o   the size and actual text of each request data item, and

o   the maximum allowed size of each response data item.

An interstation response message consists of:

o   a status code, and

o   the actual size and text of each response data item.

The cluster workstation Agent Service Process forms an interstation request message by copying the header and control information from the request block, moving the actual text of the request data items into the message, and including a specification of the maximum allowed sizes of the response data items.

After receiving the interstation response message, the cluster workstation Agent Service Process stores the status code into the request block and moves the text of the response data items into the memory areas specified for them by the request block. This transformation scheme ensures that no redundant or extraneous information is transmitted between the master workstation and cluster workstations.

OPERATIONS: PRIMITIVES

Interprocess communication management provides the operations listed below.

Check
: dequeues the message (if any) that was enqueued first at the specified exchange. Returns the status code "No message available" (14) if none are queued.

PSend
: a privileged send used by interrupt handlers. Sends the specified message to the specified exchange.

Request
: requests a system service by sending a request block to the exchange of the system service process.

Respond
: notifies a client process that the requested system service was performed by sending the request block of the client process back to the response exchange specified in the request block.

Send
: sends the specified message to the specified exchange.

Wait
: dequeues the message (if any) that was enqueued first at the specified exchange. Causes the calling process to be placed into the waiting state if no messages are enqueued.

Check

Description

The Check primitive checks whether messages are queued at the specified exchange. If messages are queued, then the message that was enqueued first is dequeued and its memory address is returned to the calling process. If no messages are queued, then status code 14 ("No message available") is returned.

The Check primitive, unlike the Wait primitive, never causes the calling process to be placed into the waiting state.

Procedural Interface

Check (exchange, ppMsgRet): ErcType

where

exchange   is the identification of the exchange to check.

ppMsgRet   is the memory address of a 4-byte field into which the memory address of the message that was enqueued first at the exchange, if any, is returned.

Request Block

Check is a Kernel primitive.

PSend

Description

The PSend primitive, a privileged Send primitive used by interrupt handlers, checks whether processes are queued at the specified exchange. If processes are queued, then the process that was enqueued first is dequeued, given the memory address of the message, and placed into the ready state.

If no processes are waiting at the exchange, then the message is queued at the exchange.

PSend uses a special pool of link blocks that are reserved at system build (see the B 20 System Programmers and Assembler Reference Manual (Part 1), form 1148699).

Procedural Interface

PSend (exchange, pMsg): ErcType

where

exchange   is the identification of the exchange to which the message is sent.

pMsg   is the memory address of the message (or a 4-byte field of information whose interpretation is agreed upon by the sending and receiving processes).

Request Block

PSend is a Kernel primitive.

Request

Description

The Request primitive requests a system service by sending a request block to the service exchange of the system service process.

A client process uses the Request primitive indirectly when it uses the procedural interface to a system service or directly when it is necessary to overlap its own execution with the performance of the service.

The Request primitive infers the appropriate service exchange by using the request code of the request block as an index into the Service Exchange Table. The use of the Service Exchange Table allows request codes to remain invariant among Operating Systems with varying organizations of system service processes. This invariance facilitates the development of filters and is critical to the transparent operation of the cluster configuration.

The client process must use the AllocExch operation (see the "Exchange Management" section) to acquire an exchange identification to place into the exchResp field of the request block.

There must not be conflicting uses of the response exchange specified in the request block; such conflict can cause malfunction of the application system that is difficult to diagnose.

Procedural Interface

Request (pRq): ErcType

where

pRq   is the memory address of the request block.

Request Block

Request is a Kernel primitive.

Respond

Description

The Respond primitive is only used by a system service process to respond to a client process. After the system service process has completed the processing of a service request, it invokes Respond to send the request block of the client process back to the response exchange specified in the request block.

The Respond primitive accepts the memory address of the request block of the client process as its only parameter; the system service process must use the same memory address as a parameter to the Respond primitive that the client process used as a parameter to the Request primitive. The exchange to which the response is directed is determined by the exchange response field of the request block.

Calls to the Respond primitive must exactly match calls to the Request primitive; that is, each Respond must answer a Request and each Request must be answered by a Respond.

A link block is reserved by the corresponding Request primitive to ensure the successful execution of the Respond primitive.

The use of the Respond primitive within an application system would cause catastrophic mismanagement of link blocks and termination of OS operation. See the discussion in the B 20 System Programmers and Assembler Reference Manual (Part 1), form 1148699, for complete explanation.

Procedural Interface

Respond (pRq): ErcType

where

pRq   is the memory address of the same request block that the system service process received from its exchange.

Request Block

Respond is a Kernel primitive.

Send

Description

The Send primitive checks whether processes are queued at the specified exchange. If processes are queued, then the process that was enqueued first is dequeued, given the memory address of the message, and placed into the ready state. If such a process has a higher priority than the calling process, it is scheduled for immediate execution and the calling process remains preempted until the higher priority process reenters the waiting state.

If no processes are waiting at the exchange, then the message is queued at the exchange.

Procedural Interface

Send (exchange, pMsg): ErcType

where

exchange   is the identification of the exchange to which the message is sent.

pMsg   is the memory address of the message (or a 4-byte field of information whose interpretation is agreed upon by the sending and receiving process).

Request Block

Send is a Kernel primitive.

Wait

Description

The Wait primitive checks whether messages are queued at the specified exchange. If messages are queued, then the message that was enqueued first is dequeued and its memory address returned to the calling process; the calling process then continues execution.

If no messages are queued, then the Process Control Block of the calling process is queued at the exchange and the process is placed into the waiting state. In the waiting state, the process stops executing and relinquishes control of the processor. The calling process remains in the waiting state until another process queues a message at the specified exchange using the Send, PSend, Request, or Respond primitives. The calling process then leaves the waiting state and is placed into the ready state. The memory address of the message queued at the exchange by the other process is returned to the calling process and it resumes execution when it becomes the highest priority ready process.

Procedural Interface

Wait (exchange, ppMsgRet): ErcType

where

exchange   is the identification of the exchange at which to wait.

ppMsgRet   is the memory address of a 4-byte field into which the memory address of the message that was enqueued first at the exchange, if any, is returned.

Request Block

Wait is a Kernel primitive.

SECTION 5
EXCHANGE MANAGEMENT

OVERVIEW

The exchange management facility supports the dynamic allocation and deallocation of exchanges. For more information about exchanges, see the "Interprocess Communication Management" section.

CONCEPTS

Exchange

An exchange is the path over which messages are communicated from process to process (or from interrupt handler to process). An exchange consists of two first-in, first-out queues: one of processes waiting for messages, the other of messages for which no process has yet waited. An exchange is referred to by a unique 16-bit integer.

Processes or messages, but not both, can be queued at an exchange at any given moment. If a process waits at an exchange at which messages are queued, then the message that was enqueued first is dequeued and its memory address given to the process; the process then continues execution. Similarly, if a message is sent to an exchange at which processes are queued, then the process that was enqueued first is dequeued, given the address of the message, and placed into the ready state.

Exchange Allocation

Exchanges are allocated in three ways:

o   Exchanges for system service processes are allocated at system build.

o   Exchanges can be dynamically allocated and deallocated using the AllocExch and DeallocExch operations.

o   When a process is created, its creator gives it a unique default response exchange. (See the "Interprocess Communication Management" section.) A process can determine the identification of its own default response exchange using the QueryDefaultRespExch operation.

In a compact system, all allocated exchanges are deallocated when the application system exits. In a system where multiple application systems can execute simultaneously, only the exchanges of an exiting application system are deallocated.

Operations and data structures for interpartition communication are described in the "Application Partition Management" section.

OPERATIONS: PROCEDURES AND SERVICES

Exchange management operations are categorized by function in Table 5-1 below.

Table 5-1. Exchange Management Operations by Function

Allocation	Deallocation
AllocExch	DeallocExch
Inquiry	
QueryDefaultRespExch	

Allocation

AllocExch   allocates an exchange.

Deallocation

DeallocExch   deallocates an exchange.

Inquiry

QueryDefaultRespExch
                     allows a process to determine
                     the identification of its own
                     default response exchange.

AllocExch

Description

The AllocExch service allocates an exchange.

Procedural Interface

AllocExch (pExchRet): ErcType

where

pExchRet   is the memory address of a word into
              which the identification of the
              allocated exchange is returned.

Request Block

sExchMax is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	40
12	reserved	6	
18	pExchRet	4	
22	sExchMax	2	2

DeallocExch

Description

The DeallocExch service deallocates an exchange.

Procedural Interface

DeallocExch (exchange): ErcType

where

exchange   is the identification of the
                exchange to deallocate. This
                identification must have been
                obtained using the AllocExch
                operation.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	41
12	exchange	2	

QueryDefaultRespExch

Description

The QueryDefaultRespExch procedure allows a process to determine the identification of its own default response exchange.

Procedural Interface

QueryDefaultRespExch (pExchRet): ErcType

where

pExchRet   is the memory address of a word into which the identification of the default response exchange of the inquiring process is returned.

Request Block

QueryDefaultRespExch is a system common procedure.

SECTION 6
MEMORY MANAGEMENT

OVERVIEW

The memory management facility supports the dynamic allocation and deallocation of areas of memory for code, data, etc., by each application system in its own partition.

Types of Memory

Two types of memory allocation are available to the application system: long-lived and short-lived. Within each application partition, long-lived memory expands upward from low memory locations, while short-lived memory expands downward from high memory locations. The OS allocates short-lived memory for application tasks.

Both long-lived and short-lived memory can be dynamically allocated and deallocated by requests to OS system services.

When the execution of an application system is terminated, the short-lived memory of its partition is automatically deallocated.

Long-lived memory is deallocated only at the explicit request of each application system. Therefore, long-lived memory is useful for passing information from an application system to a succeeding application system in the same partition.

CONCEPTS

Addressing Memory

The B20 Information Processing System has a one-megabyte address space. Each of the 1,048,576 bytes in the address space has a unique 20-bit physical memory address. However, software does not use physical memory addresses. Software identifies specific bytes of memory by using logical memory addresses.

A _logical memory address_ is a 32-bit entity consisting of a 16-bit segment base address and a 16-bit offset.

A _segment base address_ is the high-order 16-bits of the 20-bit physical memory address of a hardware segment. (The low-order 4 bits are implicitly 0.) The CS, DS, SS, and ES segment registers of the processor contain segment base addresses.

The _offset_ is the distance, in bytes, of the target location from the beginning of the hardware segment. The physical memory address of a byte is computed by multiplying the segment base address by 16 and adding the offset.

A byte of memory does not have a unique logical memory address. Rather, any of the 4096 combinations of segment base address and offset refer to the same byte of memory. Whenever the term _memory address_ is used in this Manual, it refers to logical memory address.

Segments

A _segment_ is a contiguous (usually large) area of memory that consists of an integral number of paragraphs. A _paragraph_ is 16 bytes of memory whose physical memory address is a multiple of 16.

Hardware segments can be adjacent, disjoint, partially overlapping, or completely overlapping. A physical memory location can be contained in multiple hardware segments.

Software segments are nonoverlapping hardware segments that contain single, logical entities. It is conventional to address a byte within a

software segment by using a logical memory address whose segment base address points to the first byte of the segment and whose offset is the physical memory address of the addressed byte minus the physical memory address of the first byte of the segment. This convention limits the size of a software segment to 65,536 bytes.

Code, Static Data, and Dynamic Data Segments

There are three types of software segments: code, static data, and dynamic data. Each type of segment can be either shared or nonshared.

A code segment contains only processor instructions (code) and is never modified once it is loaded into memory. This characteristic permits several processes to execute instructions from the same code segment. It also allows the virtual code segment management facility (see the section of that name) to reload code segments from the run file as needed without saving the copy of the segment previously in memory.

A data segment contains data. It can also contain code, although this is not recommended. There are no restrictions on modifying the contents of a data segment. If a data segment is shared among processes, concurrency control is the responsibility of those processes.

A static data segment is automatically loaded into memory when its containing task image is loaded.

A dynamic data segment is allocated by a request from an executing process to the memory management facility.

Code and static data segments are created by compiling and/or assembling source programs into object modules and linking the object modules into task images.

A task image is a program stored in a run file that contains code and/or static data segments. When requested, the task management facility loads the task image into memory and adjusts any logical memory addresses that exist in either code or data segments to reflect the memory address at which the task is loaded.

If the virtual code segment management facility is in use, all the static data segments, but only the <u>resident</u> code segment, are loaded into memory. The <u>nonresident</u> code segments are loaded into memory only as needed.

The Linker utility reads segments from object module files and combines them according to their segment names, class names, and directives from the user. See the B20 Systems Linker/Librarian Reference Manual, form 1148681 and B20 Systems Programmers and Assembler Reference Manual (Part 2), form 1144466.

A task image that was created by linking object modules produced by the Pascal and/or FORTRAN compilers consists of one code segment for each object module included in the link and a single static data segment. The single static data segment (DGroup) combines the static data and stack requirements of all the object modules. A task image of this form is considered standard; assembly language programmers are urged to adopt this standard unless other considerations are overriding. (The COBOL compiler and BASIC interpreter do not produce object modules.)

Memory Organization

The memory organization of an application partition in a compact system (in which application systems can be executed one at a time) differs from that of a system in which multiple application systems can be executed simultaneously.

Figure 6-1 shows the memory organization of the application partition in the compact system.

Figure 6-2 shows the memory organization of an application partition in a system in which multiple applications can be executed simultaneously. In this system, both the primary and secondary application partitions have the same memory organization.
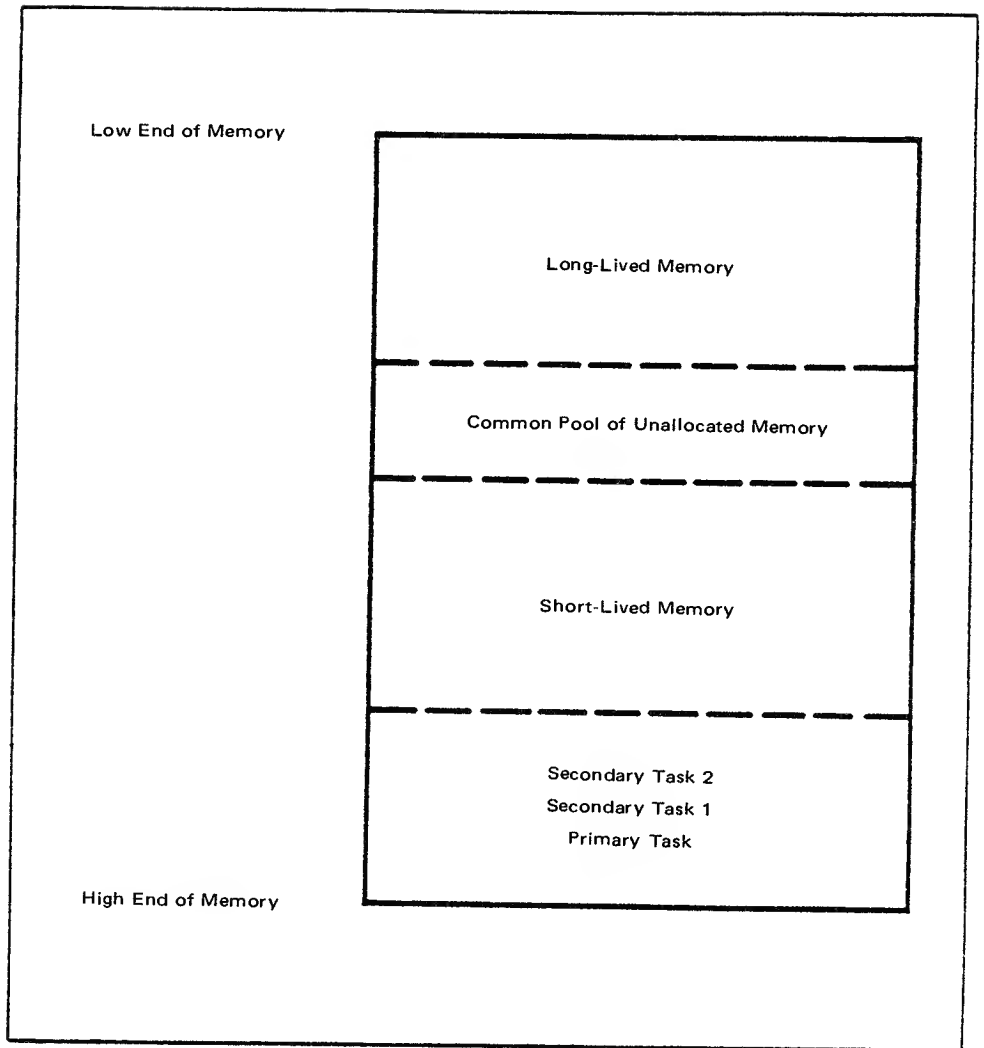
Figure 6-1. Memory Organization of the Application Partition in a Compact System

Long-Lived and Short-Lived Memory

Two types of memory allocation are available to each application system: long-lived and short-lived. Within each application partition, long-lived memory expands upward from low memory locations, while short-lived memory expands downward from high memory locations. The OS allocates short-lived memory for application tasks.

All currently unallocated memory in an application partition is in a contiguous area called the common memory pool. Memory can be allocated from both ends of the pool. There is no restriction on how much can be allocated from either end other than that the sum of the allocations cannot exceed the amount of memory available in an application partition. The QueryMemAvail operation returns the size of all available memory in an application partition.

The memory management facility of the OS allows
client processes to allocate and deallocate areas
of memory (dynamic data segments) from the common
pool in an application partition. Memory is
allocated and deallocated only on paragraph
boundaries. That is, the physical address of the
area is a multiple of 16. Because of this, areas
of memory allocated by the OS can be referenced
conveniently using the segment addressing
convention discussed previously.


```
Low End of Memory

                        ┌─────────────────────────┐
                        │   System Data Structures │
                        ├─────────────────────────┤
                        │                          │
                        │     Long-Lived Memory    │
                        │                          │
                        │ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ │
                        │                          │
                        │ Common Pool of Unallocated Memory │
                        │                          │
                        │ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ │
                        │                          │
                        │     Short-Lived Memory   │
                        │                          │
                        │ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ │
                        │                          │
                        │      Secondary Task 2    │
High End of Memory      │      Secondary Task 1    │
                        │        Primary Task      │
                        └─────────────────────────┘
```
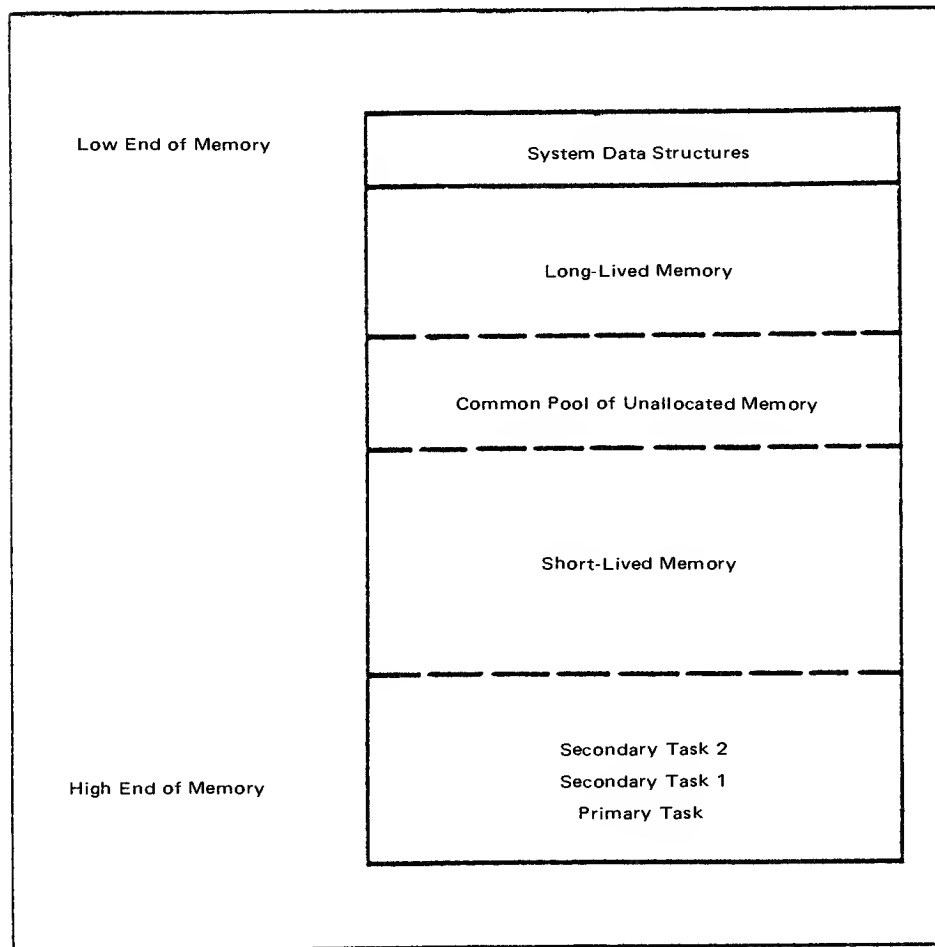
Figure 6-2. Memory Organization of an Application Partition in a System Allowing

Simultaneous Execution of Multiple Application Systems

Operations

The AllocMemoryLL and AllocMemorySL operations allocate long-lived and short-lived memory segments, respectively, in an application partition. Note, however, that the AllocAllMemorySL operation can allocate more than 65,536 bytes, and thus the entire area allocated by this operation is not necessarily addressable as a single segment.

The DeallocMemoryLL and DeallocMemorySL operations deallocate long-lived and short-lived memory segments, respectively, in an application partition. The ResetMemoryLL operation deallocates all long-lived memory in an application partition.

Deallocations

Relative to allocations from one end of an application partition's memory, deallocations must occur in exactly the opposite sequence. That is, the user must follow a last allocated, first allocated discipline when deallocating either long-lived or short-lived memory. For example, if an application system allocates short-lived memory segments A, B, and C, it then deallocates them in the order C, B, A.

Thus the motion of the borders (the dashed lines in Figures 6-1 and 6-2) of the common pool of memory in an application partition resembles the playing of an accordion: the borders converge when memory is allocated and diverge when memory is deallocated. This scheme is efficient because all unallocated memory is in a common pool and simple because the OS has to remember only the addresses of the next (long-lived and short-lived) segments to allocate, not the addresses of all allocated segments.

Long-Lived Memory Uses

The long-lived memory in an application partition is used for:

parameters passed from one application system to a succeeding application system in the same partition, and

user data that is to be processed by succeeding application systems in the same partition.

Long-lived memory allocations are returned to the common pool of unallocated memory in an application partition only upon explicit request of the application system.

Short-Lived Memory Uses

The short-lived memory in an application partition is used by the OS to contain the code and static data segments of each task. It is also allocated by application processes for use as dynamic data segments for data that is to be processed only by the current application system. Other common uses of short-lived memory are input/output buffers and the Pascal heap.

Short-lived memory allocations are returned to the common pool of unallocated memory whenever the application system is replaced (in any application partition by the Chain, ErrorExit, or Exit operations, or in the primary application partition by the key combination ACTION-FINISH). (See the "Task Management" section.)

Virtual Code Segment Management

See the "Virtual Code Segment Management" section for how tasks of an application system are handled when they require an area larger than the available physical memory in an application partition.

OPERATIONS: SERVICES

Memory management operations are categorized by function in Table 6-1.

Table 6-1. Memory Management Operations by Function

Allocation	Deallocation
AllocAllMemorySL	DeallocMemoryLL
AllocMemoryLL	DeallocMemorySL
AllocMemorySL	ResetMemoryLL

Inquiry

QueryMemAvail

Allocation

AllocAllMemorySL
> allocates the largest possible short-lived memory segment in an application partition.

AllocMemoryLL
> allocates a long-lived memory segment in an application partition.

AllocMemorySL
> allocates a short-lived memory segment in an application partition.

Deallocation

DeallocMemoryLL
> deallocates a long-lived memory segment in an application partition.

DeallocMemorySL
> deallocates a short-lived memory segment in an application partition.

ResetMemoryLL
> deallocates all long-lived memory in an application partition.

Inquiry

QueryMemAvail
> returns the size of all available memory in an application partition.

AllocAllMemorySL

Description

The AllocAllMemorySL service allocates the largest possible short-lived memory segment within an application partition.

Procedural Interface

```
AllocAllMemorySL (pcParagraphRet,
                  ppSegmentRet): ErcType
```

where

pcParagraphRet
> is the memory address of a word into which the count of bytes available (divided by 16) is returned.

ppSegmentRet is the memory address of 4 bytes into which the memory address of the allocated segment is returned. The low-order 2 bytes contain the offset, which is always 0. The high-order 2 bytes contain the segment base address of the allocated segment.

Request Block

scParagraphMax is always 2 and spSegmentMax is always 4.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	46
12	reserved	6	
18	pcParagraphRet	4	
22	scParagraphMax	2	2
24	ppSegmentRet	4	
28	spSegmentMax	4	4

AllocMemoryLL

Description

The AllocMemoryLL service allocates a long-lived memory segment of the specified size within an application partition.

Procedural Interface

AllocMemoryLL (cBytes, ppSegmentRet): ErcType

where

cBytes   is the desired segment size.

ppSegmentRet is the memory address of 4 bytes into which the memory address of the allocated segment is returned. The low-order 2 bytes contain the offset, which is always 0. The high-order 2 bytes contain the segment base address of the allocated segment.

Request Block

spSegmentMax is always 4.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	44
12	cBytes	2	
14	reserved	4	
18	ppSegmentRet	4	
22	spSegmentMax	2	4

AllocMemorySL

Description

> The AllocMemorySL service allocates a short-lived memory segment of the specified size within an application partition.

Procedural Interface

> AllocMemorySL (cBytes, ppSegmentRet): ErcType
>
> where
>
> cBytes   is the desired segment size.
>
> ppSegmentRet is the memory address of 4 bytes into which the memory address of the allocated segment is returned. The low-order 2 bytes contain the offset, which is always 0. The high-order 2 bytes contain the segment base address of the allocated segment.

Request Block

> spSegmentMax is always 4.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	42
12	cBytes	2	
14	reserved	4	
18	ppSegmentRet	4	
22	spSegmentMax	2	4

DeallocMemoryLL

Description

The DeallocMemoryLL service deallocates a long-lived memory segment of the specified size within an application partition. Segments must be deallocated in a sequence exactly opposite the one in which they were allocated (that is, last allocated, first deallocated).

Procedural Interface

DeallocMemoryLL (pSegment, cBytes): ErcType

where

pSegment   is the memory address of the segment to deallocate. The offset portion must be 0. pSegment should be the same memory address that was returned by the corresponding AllocMemoryLL operation.

cBytes   is the size (in bytes) of the segment to deallocate. cBytes should be the same value that was passed to the corresponding AllocMemoryLL operation.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	45
12	cBytes	2	
14	pSegment	4	

DeallocMemorySL

Description

The DeallocMemorySL service deallocates a short-lived memory segment of the specified size within an application partition. Segments must be deallocated in a sequence exactly opposite the one in which they were allocated (that is, last allocated, first deallocated).

Procedural Interface

DeallocMemorySL (pSegment, cBytes): ErcType

where

pSegment   is the memory address of the segment to deallocate. The offset portion must be 0. pSegment should be the same memory address that was returned by the corresponding AllocMemorySL operation.

cBytes   is the size (in bytes) of the segment to deallocate. cBytes should be the same value that was passed to the corresponding AllocMemorySL operation.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	43
12	cBytes	2	
14	pSegment	4	

QueryMemAvail

Description

The QueryMemAvail service returns the size (in 16-byte paragraphs) of all currently available memory in an application partition. Because of the way in which memory is organized, it is possible to allocate segments from available memory using both the AllocMemoryLL and AllocMemorySL operations.

Procedural Interface

QueryMemAvail (pcParagraphRet): ErcType

where

pcParagraphRet

is the memory address of a word into which the count of bytes available (divided by 16) is returned.

Request Block

scParagraphMax is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	48
12	reserved	6	
18	pcParagraphRet	4	
22	scParagraphMax	4	2

ResetMemoryLL

Description

The ResetMemoryLL service deallocates all long-lived memory within an application partition. An application system in the primary application partition should not use ResetMemoryLL unless another Executive was substituted for the Executive; this is because the Executive depends on part of the contents of long-lived memory.

Procedural Interface

ResetMemoryLL: ErcType

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	47

SECTION 7
TASK MANAGEMENT

OVERVIEW

The task management facility supports the
asynchronous execution of several loosely and/or
tightly coupled application software elements
(tasks) performing related portions of a single
application system.

An application system consists of one or more
tasks. A task consists of code, data, and one or
more processes. The code and data can be unique
to the task or shared with other tasks.

An application system can be executed in each
application partition.   Multiple application
systems can be executed simultaneously, each in
its own partition.   (See the "Application
Partition Management" section.)

Task management provides operations to (1)
replace an entire application system (all tasks
within an application partition) with a single
new task and (2) incrementally add a task to a
current application system. A task is always
loaded into the highest available memory location
within the application partition and has a single
process associated with it when it is first
loaded. Additional processes can be created
dynamically.

CONCEPTS

Application System

An application system is the name for all the tasks currently loaded in a specific application partition. These tasks can be loosely or tightly coupled, but all perform related portions of the same application system. These tasks execute asynchronously. A task can be added to an application system but not removed from it.

Task

A task is an executable program that consists of code, data, and one or more processes. The code and data can be unique to the task or shared with other tasks.

A task image is the disk-resident image of an executable program. It is created by compiling and/or assembling source language modules into object modules and linking the object modules together. A disk file that contains a task image is called a run file. A task image contains code and/or data segments.

Code and Data Segments

A code segment contains only processor instructions (code) and is never modified once it is loaded into memory. This characteristic permits several processes to execute instructions from the same code segment. It also allows the virtual code segment management facility (see the section of that name) to reload code segments from the run file as needed without saving the copy of the segment previously in memory.

A data segment contains data. It can also contain code, although this is not recommended. There are no restrictions on modifying the contents of a data segment. If a data segment is shared among processes, concurrency control is the responsibility of those processes.

A data segment that is automatically loaded into memory when its containing task image is loaded is called a static data segment, to differentiate it from a dynamic data segment. A dynamic data segment is allocated by a request from the executing process to the memory management facility.

The Linker utility reads segments from object module files and combines them according to their segment names, class names, and directives from the user. (See the B20 System Linker/Librarian Reference Manual, form 1148681, and B20 System Programmers Guide, Part 2, form 1144464.)

A task image that was created by linking object modules produced by the Pascal and/or FORTRAN compilers consists of one code segment for each object module included in the link and a single static data segment. The single static data segment (DGroup) combines the static data and stack requirements of all the object modules. A task image of this form is considered standard; assembly language programmers are urged to adopt this standard unless other considerations are overriding. (The COBOL compiler and BASIC interpreter do not produce object modules.)

Loading a Task

Loading a task consists of reading the task image into the short-lived memory of an application partition and adjusting any logical memory addresses (intersegment references) that exist in either code or data segments to reflect the memory address at which the task is loaded.

Short-lived memory is allocated from the high-address end of the common pool of unallocated memory of the application partition and is returned to the common pool whenever the application system is replaced (in any application partition by the Chain, ErrorExit, or Exit operations, or in the primary application partition by the key combination ACTION-FINISH).

If the virtual code segment management facility is in use, all the static data segments, but only the resident code segment, are loaded into memory. The nonresident code segments are loaded into memory only as needed.

Virtual code segment management is available to the primary or a secondary task of an application partition. However, a secondary task cannot be virtual if the primary task already uses virtual code segment management.

Primary tasks are those loaded by the Chain, ErrorExit, or Exit operations (see the "Task Management" section), or the LoadPrimaryTask operation (see the "Application Partition Management" section). Secondary tasks are those loaded by the task management LoadTask operation.

7-3

Exit Run File

An exit run file is a user-specified file that is loaded and activated when an application system exits. Each application partition has its own exit run file.

An application system can specify an exit run file for its partition with the SetExitRunFile operations. An application system can determine the exit run file of its partition with the QueryExitRunFile operation.

An exit run file is a primary task that can, in turn, load additional tasks into its partition with the LoadTask operation.

In the primary application partition, if no exit run file is specified, the system will malfunction and reboot itself. If the exit run file cannot be read, it displays the message "Cannot load exit run file" and a status code indicating the type of error that occurred. If the exit run file is on a floppy disk, the user can insert a floppy disk with the appropriate exit run file and the system will resume loading of the exit run file.

Operations

The task management facility provides six operations: Chain, ErrorExit, Exit, LoadTask, QueryExitRunFile, and SetExitRunFile.

Chain, ErrorExit, and Exit terminate all application processes and deallocate all short-lived memory in an application partition before loading the succeeding application system and creating a single process to execute it. In addition, ErrorExit and Exit pass an abnormal and normal status code, respectively, to the succeeding application system in the same application partition.

The LoadTask operation, in contrast, preserves all current application processes and short-lived memory allocations in the application partition while loading and activating an additional task and creating an additional process to execute it.

The SetExitRunFile operation establishes a new exit run file for an application partition. The QueryExitRunFile operation returns the name, password, and priority of the exit run file of an application partition.

OPERATIONS: PROCEDURES AND SERVICES

Task management provides the operations listed below.

Chain   replaces the current application system in an application partition with the specified run file.

ErrorExit   terminates the current application system in an application partition and passes an abnormal status code to the exit run file.

Exit   terminates the current application system in an application partition and passes a normal status code to the exit run file.

LoadTask   loads and activates an additional task as part of the current application system in an application partition.

QueryExitRunFile

                returns the name, password, and priority of the exit run file of an application partition.

SetExitRunFile

                establishes a new exit run file for an application partition.

Chain

Description

The Chain service replaces a current application system with a specified run file. Chain returns control to the calling process only if an error condition is detected.

Chain:

1. Verifies that the specified run file exists, that it can be opened for Read using the password provided, that it contains a valid task image, and that the task image fits in the application partition.

2. Places the status code in the Application System Control Block of the application partition.

3. Disconnects interrupt handlers of the application partition and terminates all processes of the application partition.

4. Terminates keyboard (primary application partition only), timer, and communications requests, and waits until all disk and printer input/output activity has ceased.

5. In the primary application partition only, resets the keyboard to character mode. Discards the content of the type-ahead buffer if the keyboard was in unencoded mode and/or the status code is nonzero.

6. In the primary application partition only, reenables the ACTION-FINISH feature and discards the action code (if any).

7. In the primary application partition only, closes the submit or recording file if the status code is nonzero.

8. Closes all files opened for the application partition except those marked long-lived (by the OpenFileLL or SetFhLongevity operations; see the "File Management" section).

9. Releases for reuse all application partition memory that was allocated as short-lived.

10. Allocates a short-lived memory segment in the application partition that is large enough to contain the task image from the specified run file.

11. Reads the task image from the run file into the application partition.

12. Relocates all intersegment reference to accommodate the memory address at which the task image is loaded.

13. Creates a process to be scheduled at the specified priority. The initial values loaded into the segment registers (CS, DS, SS, ES), the Stack Pointer (SP), and the Instruction Pointer (IP) are derived from information in the run-file header.

Chain has no effect on the allocation of long-lived memory.

If the task requires virtual code segment management, the run file is left open to accommodate code swapping. The file handle of the open run file is placed in the Application System Control Block of the application partition.

Procedural Interface

```
Chain (pbFileSpec, cbFileSpec, pbPassword,
     cbPassword, priority, ercTermination,
     fDebug): ErcType
```

where

pbFileSpec
cbFileSpec   describe a character string of the form {node} [volname] <dirname> filename.

pbPassword
cbPassword   describe either the volume, directory, or file password that authorizes access to the specified file.

priority   is the priority (0-254, with 0 the highest) at which to schedule the newly created process for execution.

ercTermination

> is a 16-bit status code to be placed in the Application System Control Block of the application partition for examination by the run file. In the primary application partition only, a nonzero status code causes the content of the type-ahead buffer to be discarded and the submit or recording file to be closed.

fDebug

> indicates whether the run file is to be debugged. TRUE indicates it is to be debugged and therefore not scheduled for execution; FALSE indicates it is to be scheduled for execution. If fDebug is TRUE, then the Debugger is entered automatically as soon as the task image is loaded into the application partition.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	28
12	priority	2	
14	ercTermination	2	
16	fDebug	2	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	

ErrorExit

Description

The ErrorExit procedure terminates the current application system and passes an abnormal status code to the specified exit run file. ErrorExit never returns to the calling process.

ErrorExit is exactly like the Exit operation except that the status code in ErrorExit is explicit.

ErrorExit:

1. Verifies that the specified exit run file exists, that it contains a valid task image, and that the task image fits in the application partition memory (the OS terminates if this verification fails.)

2. Places the specified abnormal status code in the Application System Control Block of the application partition.

3. Disconnects interrupt handlers of the application partition and terminates all processes of the application partition.

4. Terminates keyboard (primary application partition only), timer, and communications requests, and waits until all disk and printer input/output activity has ceased.

5. In the primary application only, resets the keyboard to character mode. Discards the content of the type-ahead buffer if the keyboard was in unencoded mode and/or the status code is nonzero.

6. In the primary application partition only, reenables the ACTION-FINISH feature and discards the action code (if any).

7. In the primary application partition only, closes the submit or recording file if the status code is nonzero.

8. Closes all files opened for the application partition except those marked long-lived (by the OpenFileLL or SetFhLongevity operations; see the "File Management" section).

9. Releases for reuse all application partition memory that was allocated as short-lived.

10. Allocates a short-lived memory segment large enough to contain the task image from the specified exit run file. If sufficient application partition memory to load the exit run file cannot otherwise be allocated, then long-lived memory is reset (that is, released to the common pool of unallocated memory) before the exit run file is loaded.

11. Reads the task image from the run file into the application partition.

12. Relocates all intersegment references to accommodate the memory address at which the task image is loaded.

13. Creates a process to be scheduled at the default priority. The initial values loaded into the segment registers (CS, DS, SS, ES), the Stack Pointer (SP), and the Instruction Pointer (IP) are derived from information in the run-file header.

ErrorExit has no effect on the allocation of long-lived memory except as noted in step 10 above. If necessary, the exit run file is left open to accommodate code swapping of the exit run file. The file handle of the open run file is placed in the Application System Control Block of the application partition.

Procedural Interface

Call ErrorExit (ercTermination)

ercTermination
    is a 16-bit status code to be placed in the Application System Control Block of the application partition for examination by the exit run file. In the primary application partition only, a nonzero status code causes the content of the type-ahead buffer to be discarded and the submit or recording file to be closed.

Request Block

ErrorExit is a system common procedure.

Exit

Description

The Exit procedure terminates the current application system and passes a normal status code to the specified exit run file. Exit never returns to the calling process.

Exit is exactly like the ErrorExit operation except that the status code in Exit is implicit. That is,

    Call Exit
is equivalent to:
    Call ErrorExit (0)

Exit:

1. Terminates the current application system.

2. Places a normal successful status code (0) in the Application System Control Block of the application partition.

3. Closes all files opened for the specific application partition except those marked long-lived (by the OpenFileLL or SetFhLongevity operations; see the "File Management" section).

4. Invokes the exit run file of the application partition.

Procedural Interface

    Call Exit

Request Block

Exit is a system common procedure.

LoadTask

Description

The LoadTask service loads and activates an additional task as part of the current application system in the application partition.

LoadTask:

1. Verifies that the file handle specifies a run file that contains a valid task image and that the task image fits in the application partition memory.

2. Allocates a short-lived memory segment large enough to contain the task image from the specified run file.

3. Reads the task image from the run file into the application partition.

4. Relocates all intersegment references to accommodate the memory address at which the task image is loaded.

5. Creates a process to be scheduled at the specified priority. The initial values loaded into the segment registers (CS, DS, SS, ES), the Stack Pointer (SP), and the Instruction Pointer (IP) are derived from information in the run-file header.

Procedural Interface

LoadTask (fh, priority, fDebug): ErcType

where

fh   is the file handle of a run file that has been opened by the calling process.

priority   is the priority (0-254, with 0 the highest) at which to schedule the newly created process for execution. A value of 255 requests that a process not be created. This permits the loading of a task image that is executed by calling the procedures in it from another process.

fDebug   indicates whether the task is to be
                debugged.   TRUE indicates it is to
                be   debugged   and   therefore   not
                scheduled   for   execution;   FALSE
                indicates it is to be scheduled for
                execution.   In   contrast   to   its
                meaning   in   the   Chain   operation,
                setting   fDebug   to   TRUE   does   not
                automatically activate the Debugger.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	29
12	fh	2	
14	priority	2	
16	fDebug	2	

QueryExitRunFile

Description

The QueryExitRunFile service returns the name, password, and priority of the exit run file of the application partition.

Procedural Interface

```
QueryExitRunFile (pbExitRunFileRet,
                  cbExitRunFileRet,
                  pbPasswordRet, cbPasswordRet,
                  pbPriorityRet): ErcType
```

pbExitRunFileRet
cbExitRunFileRet
> define the memory area into which the exit run file specification is returned. The first byte of the returned information is the size of the exit run file specification.

pbPasswordRet
cbPasswordRet
> define the memory area into which the password for the exit run file is returned. The first byte of the returned information is the size of the password.

pbPriorityRet
> is the memory address of the word into which the priority of the exit run file is returned.

Request Block

cbPriorityRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	3
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	187
12	reserved	6	
18	pbExitRunFileRet	4	
22	cbExitRunFileRet	2	
24	pbPasswordRet	4	
28	cbPasswordRet	2	
30	pbPriorityRet	4	
34	cbPriorityRet	2	2

SetExitRunFile

Description

The SetExitRunFile service establishes a new exit run file for the application partition in which the calling process is executing.

Procedural Interface

SetExitRunFile (pbExitRunFile, cbExitRunFile,
                pbPassword, cbPassword,
                priority): ErcType

where

pbExitRunFile
cbExitRunFile

describe a character string of the form {node}[volname]<dirname>file-name that specifies the run file to be loaded into the application partition when an Exit request is issued by the current task.

pbPassword

cbPassword   describe the volume, directory, or file password that authorizes access to the specified file.

priority   is the priority (10-254, with 10 the highest) at which the newly created process is scheduled for execution.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqcode	2	186
12	priority	2	
14	reserved	4	
18	pbExitRunFile	4	
22	cbExitRunFile	2	
24	pbPassword	4	
28	cbPassword	2	

SECTION 8
VIRTUAL CODE SEGMENT MANAGEMENT

OVERVIEW

The virtual code segment management facility permits the execution of application systems that exceed the size of physical memory of an application partition. This is accomplished through the use of the virtual memory technique.

Virtual code segment management is available to the primary or a secondary task of an application partition. However, a secondary task cannot be virtual if the primary task already uses virtual code segment management.

Primary tasks are those loaded by the Chain, ErrorExit, or Exit operations (see the "Task Management" section), or the LoadPrimaryTask operation (see the "Application Partition Management" section).

Secondary tasks are those loaded by the task management LoadTask operation.

CONCEPTS

Virtual Memory

Virtual memory is a technique that makes the apparent size of memory of an application partition (from the perspective of the application programmer) greater than its actual physical size. This improves the efficiency of memory usage by allowing disk storage, as well as the main memory of an application partition, to be used to contain parts of the current application system.

Two popular implementations of virtual memory are segment swapping and page swapping. (The use of program overlays is not considered virtual memory because it is not transparent to the application programmer.)

Virtual Code Segment Swapping

The Operating System supports virtual code segment swapping. Each task is divided into variable-length code segments that reside on disk in a run file. As the task executes, only those code segments that are required at a particular time actually reside in the main memory of an application partition; the other code segments remain on disk until they, in turn, are required.

When a particular code segment in the memory of an application partition is no longer needed, it is overlaid by another code segment. This can be done because all code segments produced by B20 compilers (and by assembler code that is written according to a simple set of guidelines; see the B20 System Programmers and Assembler Reference Manual (Part 2), form 1144466) are reentrant.

When the particular code segment is required again, it is simply reread from the run file. Since code segments are never modified, they can always be read directly from the run file into which the Linker wrote them.

Virtual Code Segment Swapping Versus Page Swapping

OS virtual code segment swapping differs from the page swapping of other systems in two significant ways:

o   Since only code, not data, segments are moved from disk to the main memory of an application partition, it is never necessary to allocate a swapping file or to write segments back to disk.

o   A code segment is a variable-length logical entity, not a fixed-length physical entity. A code segment contains one or more complete procedures.

Using the Virtual Code Segment Management Facility

There are two steps to using virtual code segment management:

o   initializing the virtual code segment management facility, and

o   specifying to the Linker the desired grouping of object modules into code segments.

Initializing

The swap buffer is an overlay area in the memory of an application partition. It is used to contain all nonresident code segments. It must be allocated either dynamically using the AllocMemorySL operation (see the "Memory Management" section) or statically configured into the task. The swap buffer is commonly allocated dynamically so that its size can be determined by the amount of memory available in the partition.

The InitOverlays object module procedure must be called before any procedure in a nonresident (virtual) code segment is called.

The arguments to the InitOverlays operation are the memory address and the size of the swap buffer. This buffer must be large enough to contain the largest nonresident code segment. A larger buffer permits more code segments to be kept in the main memory of an application partition and improves system performance.

After the virtual code segment management
facility is initialized, no further explicit
reference must be made to the swap buffer; the
facility automatically allocates the memory in
the swap buffer to code segments as they are read
in.

Linking

When linking a task to use the virtual code
segment management facility, the desired grouping
of object modules into code segments must be
specified to the Linker.

No restrictions are placed on the ability of
procedures to call other procedures in any code
segment to any degree of nesting or recursion.
Note, however, that the performance of an
application system is substantially improved if
some care is exercised in the grouping of
procedures into object modules and object modules
into code segments. (See the B20 Systems
Linker/Librarian Reference Manual, form 1148681,
for more information about the Linker utility.)

Using Overlays

Programs that use overlays have two parts:
resident and overlaid.

The resident part contains resident code and
data. It must contain the main program and the
call to the InitOverlays operation.

The overlaid part contains one or more overlays.
Each overlay corresponds to one or more code
segments. Only code segments can be overlaid.
All other segments must remain memory-resident.

The Linker identifies code segments by the class
name CODE. This is set automatically by FORTRAN
and Pascal but must be set explicitly when using
assembly language.

Normally, a code segment is generated by a single
compilation and is contained in one object
module. However, the Linker can combine code
segments in any number of object modules into a
single code segment.

OPERATIONS: PROCEDURES

When using overlays:

The maximum size of the resident code is equal to the total memory in the B 20 minus the used memory (ie. Operating System, system services and user programs).

The maximum number of overlays is 256.

Any procedures called before the overlay area is initialized must be in the resident code.

The SwapAl, Swap0, Swapl, Swap2, and ComSub object modules in the OS library must be in the resident code.

All callers of the LockIn and LockOut operations in ComSub (for example, SamCop) must be in the resident code.

Virtual code segment management provides the operation listed below.

GetCParasOvlyZone	returns the size of the swap buffer measured in paragraphs.
InitLargeOverlays	is identical to InitOverlays service with the exception that the user describes the length of the swap buffer as a count of paragraphs instead of bytes.
InitOverlays	initializes the virtual code segment management facility.
MakeRecentlyUsed	is called from within an overlay to prevent that overlay from being inadvertently swapped out.
ReInitLargeOverlays	is identical to the ReInitOverlays with the exception that the user describes the length of the swap buffer as a count of paragraphs instead of bytes.
ReInitOverlays	allows the user to change the size of the swap buffer to recover memory or extend the swap buffer for better performance.

GetCParasOvlyZone

The GetCParasOvlyZone service returns the size of the swap buffer measured in paragraphs. A paragraph is 16 bytes.

Procedural Interface

GetCParasOvlyZone: WORD

where the returned word contains the size of the swap buffer measured in paragraphs.

Request Block

GetCParasOVlyZone is an object module procedure.

InitLargeOverlays

The InitLargeOverlays service is identical to the InitOverlays service with the exception that the user describes the length of the swap buffer as a count of paragraphs instead of bytes. A paragraph equals 16 bytes.

Procedural Interface

InitLargeOverlays (pSwapBuffer, cParasSwapBuffer)

where

pSwapBuffer   is the memory address of the first byte of the swap buffer. The buffer must be word-aligned.

cParasSwapBuffer   is the size of the swap buffer in paragraphs. A paragraph is 16 bytes. The buffer must be large enough to contain the largest non- resident code segment.

Request Block

InitLargeOverlays is an object module procedure.

InitOverlays

Description

The InitOverlays procedure initializes the virtual code segment management facility. InitOverlays is called once at the beginning of a task. It must be included in the resident code of a task and must be called before any procedure in a nonresident (virtual) code segment is called.

Procedural Interface

InitOverlays (pSwapBuffer, sSwapBuffer)

where

pSwapBuffer   is the memory address of the first byte of the swap buffer. The buffer must be word-aligned.

sSwapBuffer   is the size of the swap buffer. The buffer must be large enough to contain the largest nonresident code segment.

Request Block

InitOverlays is an object module procedure.

MakeRecentlyUsed

The MakeRecentlyUsed service is called from within an overlay to prevent that overlay from being inadvertently swapped out. The default replacement algorithm of the Virtual Code facility will swap out overlays based on age in memory. MakeRecentlyUsed over rides this default by making an overlay appear to have zero age regardless of when it was swapped in. When MakeRecentlyUsed is called by an overlay, it will only be replaced in memory if there is insufficient room for itself and for the next overlay called. In this way the user may dictate to the virtual code facility that a specific overlay should remain in memory if possible.

Procedural Interface

MakeRecentlyUsed

with no arguments

Request Block

MakeRecentlyUsed is an object module procedure.

ReInitLargeOverlays

The ReInitLargeOverlays service is identical to the ReInitOverlays service documented above with the exception that the user describes the length of the swap buffer as a count of paragraphs instead of bytes. A paragraph equals 16 bytes.

Procedural Interface

ReInitLargeOverlays (cParasSwapBuffer)

where

cParasSwapBuffer   is the size of the swap
                    buffer in paragraphs. A
                    paragraph is 16 bytes. The
                    buffer must be large enough to
                    contain the largest
                    nonresident code segment.

Request Block

ReInitLargeOverlays is an object module procedure.

ReInitOverlays

The ReInitOverlays service allows the user to change the size of the swap buffer to recover memory or extend the swap buffer for better performance. The swap buffer size can only be changed by adding or subtracting memory from the high memory side of the buffer. Remember that the size of the swap buffer must always follow the constraints previously described.

Procedural Interface

ReInitOverlays (sSwapBuffer)

where

sSwapBuffer   is the size of the swap buffer. The buffer must be large enough to contain the largest nonresident code segment.

Request Block

ReInitOverlays is an object module procedure.

SECTION 9
PARAMETER MANAGEMENT

OVERVIEW

The parameter management facility provides a structured mechanism for passing limited information from one application system to its successor within the same partition.

Application systems that pass parameters include, for example, the B20 Executive in the primary application partition, or the batch manager in any application partition.

Forms-Oriented Interface

The B20 Information Processing System supports and encourages the use of forms-oriented interfaces for workstation operators.

The B20 Executive is an example of a forms-oriented interface. The operator types a command name and presses the RETURN key; the Executive responds with the command form appropriate to it. (See the B20 System Executive Reference Manual, form 1144474 for details about this type of interface.)

For example, if the operator types Delete and presses RETURN, the following form appears:

```
Delete
   File list        _____
   [Confirm each?]  _____
```

The operator enters data into the fields of the form and also corrects typing errors by modifying the data. The operator, when satisfied with the contents of the fields, presses the GO key.

Parameters

Note that the Delete command takes two kinds of parameters: a parameter and a list of subparameters. A parameter consists of zero or more subparameters. A subparameter typically consists of an arbitrary sequence of characters not including a space. (For Executive parameters, see the "Parameters in a Command Form" section of the B20 System Executive Reference Manual, form 1144474.)

Organizing Parameters: Variable-Length Parameter Block

Continuing the previous example, after the operator has pressed GO, the Executive organizes the operator's data to simplify Delete's extraction of the parameters. The organized data is stored in the Variable-Length Parameter Block.

The Variable-Length Parameter Block (VLPB) is a formal structure used by the Executive and batch manager to communicate parameters to application systems. The VLPB is created in the long-lived memory of an application partition, and its memory address is stored in the Application System Control Block (ASCB) of the application partition.

An ASCB in each application partition communicates parameters and other information between application systems within its partition.

The VLPB and the parameter passing services of the Executive and batch manager are applicable to any application system on a B20 system.

A common case is an application system to be invoked from the Executive. When implementing such an application system, the user decides on a command name, the captions for the fields of the command form, and the corresponding message that appears when the operator presses the HELP key. This information is supplied to the Executive using the New Command command (as described in the B20 System Executive Reference Manual, form 1144474).

Another common case is an application system to be invoked from the batch manager. When implementing such an application system, the user creates a batch job control language file.

CONCEPTS

Parameter and Subparameter

A <u>parameter</u> consists of zero or more subparameters. A <u>subparameter</u> typically consists of an arbitrary sequence of characters not including a space. For example, the parameter:

    1 abc Work.Fri

contains three subparameters: 1, abc, and Work.Fri.

A space is embedded in a subparameter by including the entire subparameter in single quotes. For example, the parameter:

    '1 abc' Work.Fri

contains two subparameters: 1 abc, and Work.Fri.

Variable-Length Parameter Block

The Variable-Length Parameter Block (VLPB) is a formal structure used by the Executive or batch manager to communicate parameters to application systems in an application partition. The VLPB is created in the long-lived memory of an application partition; its memory address is stored in the pVLPB field of the Application System Control Block (see below). The application system gets its parameters from the VLPB using three operations: CParams, CSubParams, and RgParam.

The CParams operation returns the number of parameters stored in the VLPB, that is, the number of fields in the command form.

The CSubParams operation returns the number of subparameters stored in the VLPB for a specified parameter, that is, the number of subparameters the operator entered in a specified field of the command form.

The RgParam operation provides access to the parameters stored in the VLPB.

Four object module procedures support the creation of a VLPB: RgParamInit, RgParamSetElt-Next, RgParamSetListStart, and RgParamSetSimple.

The VLPB is a self-describing, two-dimensional
array of character strings. Each element of the
array rgSdoParam is a pair (ob, cb) of words,
where ob is the offset within the VLPB of the
corresponding row of the two-dimensional array,
and cb is the number of bytes occupied by the
row. The strings that make up a row are prefixed
with a 1-byte count and packed together without
padding.

The format of the VLPB is shown in Table 9-1
below.

Table 9-1. Variable-Length Parameter Block

Offset	Field	Size (bytes)
0	sVarParams	2
2	ibFirstFree	2
4	cParams	2
6	rgSdoParam (cParams + 1)	4*(cParams + 1)

Application System Control Block

An Application System Control Block (ASCB) in
each application partition communicates
parameters, the termination code, and other
information between application systems within
its partition.

The address of the ASCB is obtained through the
GetASCB operation.

The format of the ASCB is shown in Table 9-2.

Table 9-2. Application System Control Block

Offset	Field	Size (bytes)	Description
0	fhSwapFile	2	Set by the Chain operation. (See the "Task Management" section.) If the primary task is virtual, then fhSwapFile is the file handle of its run file; otherwise, fhSwapFile is set to 0FFFFh.
2	pVLPB	4	Memory address of the VLPB in the long-lived memory of an application partition.
6	fExecScreen	1	Set to FALSE by the ResetVideo operation (see the "Video Display Management" section) and to TRUE by the Executive. If fExecScreen is FALSE when the B20 Executive is loaded, it reinitializes the video subsystem.
7	fChkBoot	1	Set to FALSE during OS initialization and to TRUE by the Executive.
8	ercRet	2	The Chain operation writes its ercTermination parameter into this word.

Table 9-2. Application System Control Block (Cont.)

Offset	Field	Size (bytes)	Description
10	pbMsgRet	4	
14	cbMsgRet	2	pbMsgRet and cbMsgRet can be set by an application system to describe a string of text located in long-lived memory. When the Executive is loaded, this text appears on the video display.
16	reserved	8	
22	fTermination	1	Set to TRUE when a user tries to ACTION-FINISH an application system when ACTION-FINISH is disabled; or when an application system tries to terminate the task in a locked secondary partition. This is set to FALSE when a task replaces the old task in the partition
23	fVacate	1	Set to TRUE when a user or an application system tries to vacate the task in a locked secondary partition. This is set to FALSE when a task replaces the old task in the partition.
24	oLastTask	2	Offset of the last task loaded.

Table 9-2. Application System Control Block (Cont.)

Offset	Field	Size (Bytes)	Description
26	fExecFont	1	Set to FALSE by OS when the font is changed. When the Executive finds fExec-font set to FALSE, it reloads the font and sets fExecfont to TRUE.
27	bActionCode	1	Contains the last action code detected by the keyboard process (not including ACTION-A, -B, and FINISH codes).
28	cParMemArray	2	The size of the memory array (in 16-byte paragraphs) of the primary task when loaded.
30	reserved	34	

Table 9-2. Application System Control Block (Cont.)

Offset	Field	Size (Bytes)	Description
64	sbUserName	31	The name of the current user (the first byte of the string is the length of the name).
95	sbPassword	13	The password the user gave when signing on (for accessing the user configuration file).
108	sbCmdFile	79	The name of the user's Executive command file.

OPERATIONS: PROCEDURES

Parameter management operations are categorized by function in Table 9-3 below.

Table 9-3. Parameter Management Operations by Function

Retrieval	Creation
CParams	RgParamInit
CSubParams	RgParamSetEltNext
GetpASCB	RgParamSetListStart
RgParam	RgParamSetSimple

Retrieval

CParams	returns the number of parameters stored in the Variable-Length Parameter Block.
CSubParams	returns the number of subparameters stored in the Variable-Length Parameter Block for a specified parameter.
GetpASCB	returns the address of the Application System Control Block in an application partition.
RgParam	provides access to the parameters stored in the Variable-Length Parameter Block.

Creation

RgParamInit	initializes the specified memory to be the Variable-Length Parameter Block.
RgParamSetEltNext	creates an additional subparameter of the current parameter in the Variable-Length Parameter Block.
RgParamSetListStart	initiates the creation of a parameter with multiple subparameters.
RgParamSetSimple	creates a parameter with one subparameter.

CParams

Description

The CParams procedure returns the number of parameters stored in the Variable-Length Parameter Block, that is, the number of fields in the command form.

Note that the B20 Executive passes the name of the command as parameter zero.

Procedural Interface

CParams: WORD

Request Block

CParams is an object module procedure.

CSubParams

Description

The CSubParams procedure returns the number of subparameters stored in the Variable-Length Parameter Block for a specified parameter, that is, the number of subparameters the operator entered in a specified field of the form.

Procedural Interface

CSubParams (iParam): WORD

where

iParam   is the index of the parameter.

Request Block

CSubParams is an object module procedure.

GetpASCB

Description

        The GetpASCB procedure returns the address of the
Application System Control Block (ASCB) of the
application partition in which the application
system is executing.

Procedural Interface

        GetpASCB (ppASCBRet): ErcType

        where

        ppASCBRet   is the memory address of a pointer
                         that is returned with the address to
                         the ASCB.

Request Block

        GetpASCB is a system common procedure.

RgParam

Description

The RgParam procedure provides access to the parameters stored in the Variable-Length Parameter Block. Each RgParam invocation returns the memory address and size of a subparameter. Note that the Executive stores the command name used to invoke the application system in RgParam (0,0).

Procedural Interface

RgParam (iParam, jParam, pSdRet): ErcType

where

iParam   is the index of the parameter.

jParam   is the index of the subparameter.

pSdRet   is the location of a 6-byte block of memory. The memory address of the subparameter is returned in the first 4 bytes, and its size is stored in the last 2 bytes.

Request Block

RgParam is an object module procedure.

RgParamInit

Description

The RgParamInit procedure initializes the specified memory to be the Variable-Length Parameter Block. If the block of memory is not large enough, RgParamInit attempts to increase its size by allocating additional long-lived memory. This attempt succeeds only if the block of memory is at the top of the long-lived memory of an application partition.

Procedural Interface

```
RgParamInit (pVarParams, sVarParams,
             iParamMax): ErcType
```

where

pVarParams
sVarParams   describe the block of memory to be used for the Variable-Length Parameter Block. If sVarParams is 0, the current Variable-Length Parameter Block is reinitialized.

iParamMax   is one less than the number of parameters to be recorded.

Request Block

RgParamInit is an object module procedure.

RgParamSetEltNext

Description

The RgParamSetEltNext procedure creates an additional subparameter of the current parameter in the Variable-Length Parameter Block. The invocation of RgParamSetEltNext must immediately follow the invocation of either the RgParamSet-ListStart or RgParamSetEltNext procedure.

If the Variable-Length Parameter Block is not large enough to accommodate this subparameter, it is compacted and an attempt made to extend it by allocating additional long-lived memory. This attempt succeeds only if the Variable-Length Parameter Block is at the top of the long-lived memory of an application partition.

Procedural Interface

RgParamSetEltNext (pSd): ErcType

where

pSd   is the location of a 6-byte block of memory, the first 4 bytes of which contain the memory address of the string to be used and the last 2 bytes of which contain the string's length.

Request Block

RgParamSetEltNext is an object module procedure.

RgParamSetListStart

Description

The RgParamSetListStart procedure initiates the creation of a parameter with multiple subparameters. The RgParamSetEltNext procedure, which must be called immediately following an invocation of RgParamSetListStart, creates a subparameter. If the parameter already exists, all its old subparameters are destroyed and the memory they occupied reused.

Procedural Interface

RgParamSetListStart (iParam): ErcType

where

iParam   is the index of the parameter.

Request Block

RgParamSetListStart   is   an   object   module procedure.

RgParamSetSimple

Description

The RgParamSetSimple procedure creates a parameter with one subparameter. If the parameter already exists, all its old subparameters are destroyed and the memory they occupied reused.

If the Variable-Length Parameter Block is not large enough to accommodate this parameter, it is compacted and an attempt made to extend it by allocating additional long-lived memory. This attempt succeeds only if the Variable-Length Parameter Block is at the top of the long-lived memory of an application partition.

Procedural Interface

RgParamSetSimple (iParam, pSd): ErcType

where

iParam   is the index of the parameter.

pSd   is the location of a 6-byte block of memory, the first 4 bytes of which contain the memory address of the string to be used and the last 2 bytes of which contain the string's length.

Request Block

RgParamSetSimple is an object module procedure.

SECTION 10
APPLICATION PARTITION MANAGEMENT

OVERVIEW

The application partition management facility supports the simultaneous execution of several application systems, each in its own partition. An interactive application system can be executing in one partition while noninteractive application systems are executing in other partitions.

Each application system can load and activate any number of tasks within its partition. Any number of processes can execute the code in each task. Each application system is completely independent of the others, yet can communicate with application systems in other partitions.

CONCEPTS

Types of Partitions

The memory of a system consists of two types of partitions:

system partitions, which are loaded with the operating system (OS) and dynamically installed system services, and

application partitions, each of which can be loaded with an application system.

When a system is initialized, the OS is loaded into the system partition at the low-address end of memory. Dynamically installed system services are loaded into an extended system partition located at the high-address end of memory. All remaining memory is defined as a single application partition called the primary application partition. (See Figure 10-1 below.)

When new partitions are created, they are placed at the high-address end of the existing application partition and are called secondary application partitions. The remaining memory is defined as the primary application partition. (See Figure 10-2 below.)

Types of Application Partitions

Primary Application Partitions

The primary application partition is for interactive programs that use the keyboard and video display to interact with the user. Such partitions can be loaded with interactive programs chosen by the user, such as the Editor, Word Processor, or Terminal Emulators.

Secondary Application Partitions

Secondary application partitions are for noninteractive applications. Such partitions can be loaded with user applications, the batch manager, and system services (such as the printer spooler, ISAM, or a remote job entry).

Application systems executing in secondary application partitions under control of the batch manager have their keyboard input and video output automatically redirected to System Input (SysIn) and System Output (SysOut) facilities.
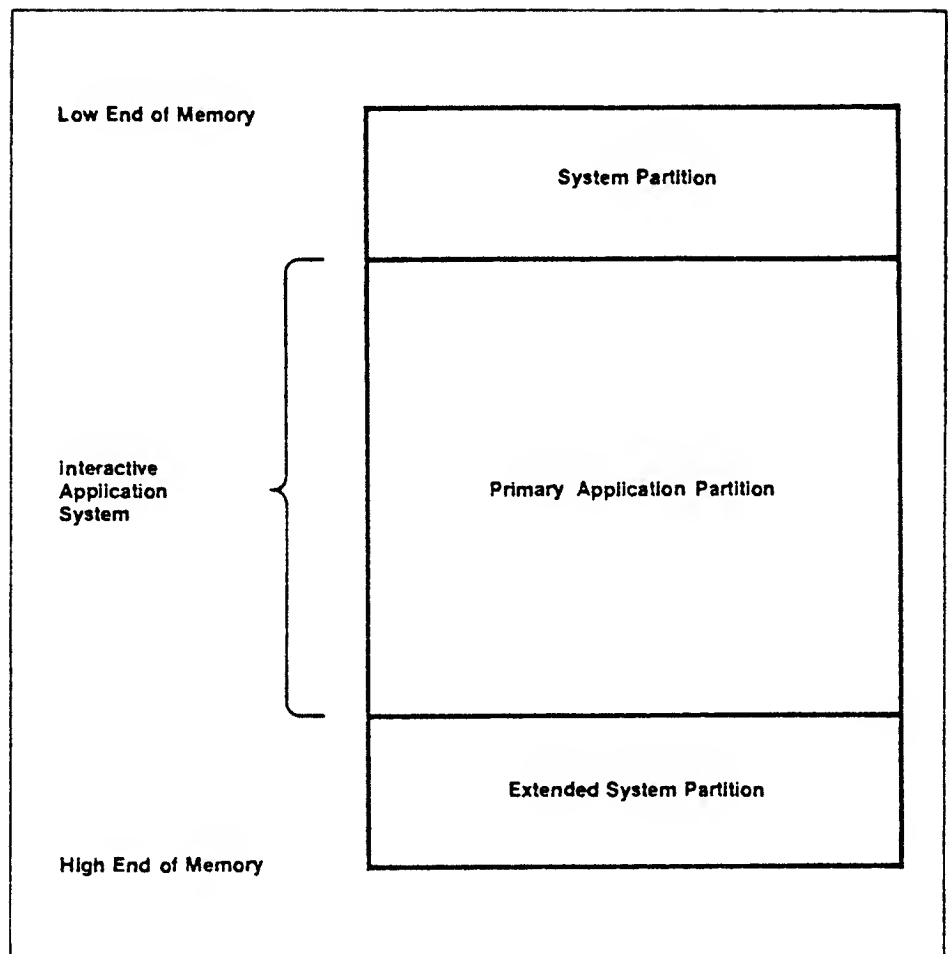
Figure 10-1. Memory Organization Without Secondary Application Partitions

Dynamic Control of Application Partitions

Application partitions are dynamically controlled through utilities (described in the B 20 System Software Operation Guide, form 1148772) or operations (described in this Manual).

The operations described in this section control processing in secondary application partitions. Operations described elsewhere in this Manual apply to all application partitions, unless otherwise noted.
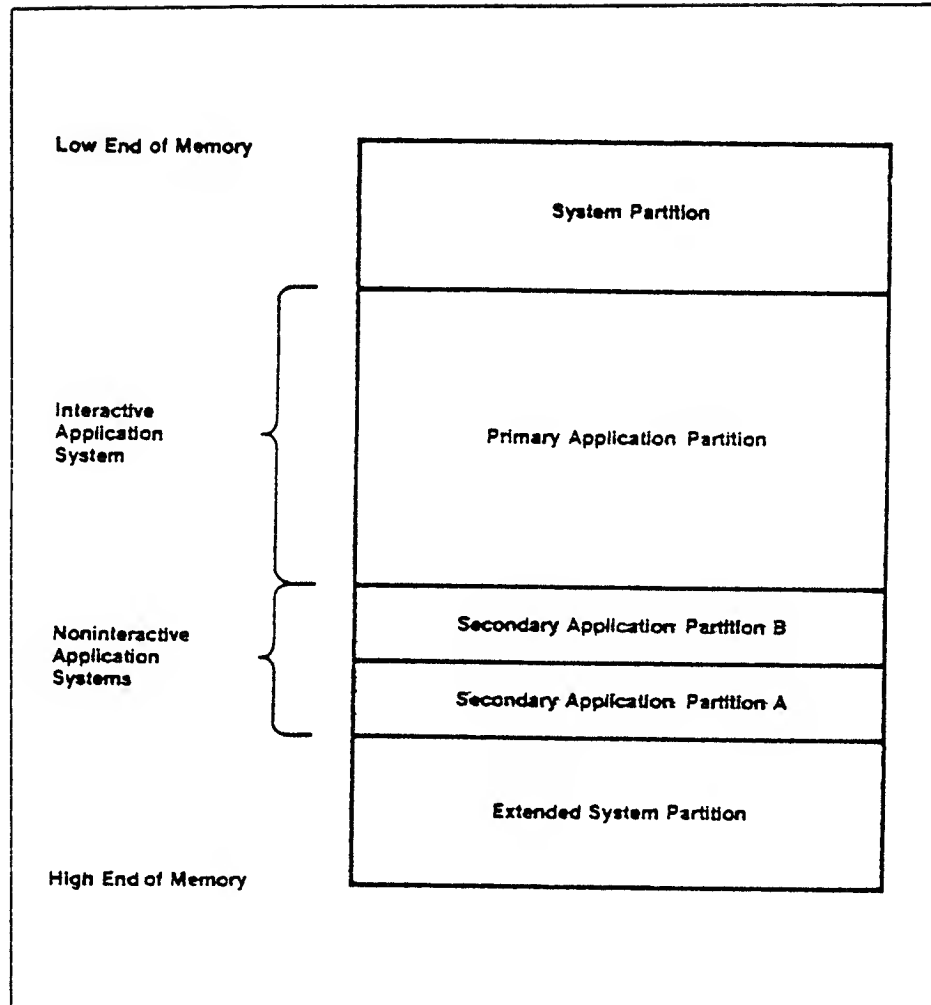
Figure 10-2. Memory Organization With Secondary Application Partitions

Memory Organization of Application Partitions

The memory of application partitions is organized as shown in Figure 10-3. The entities in the partition are:

> system data structures describing the partition and its current application system, and

> primary and secondary tasks that make up the current application system.

A process executing in an application partition
can allocate and deallocate the memory of its own
partition. Long-lived memory is allocated from
the low-address end, and short-lived memory from
the high-address end of the partition. A process
cannot allocate or deallocate memory in other
partitions.

Low End of Memory

System Data Structures

Long-Lived Memory

Common Pool of Unallocated Memory

Short-Lived Memory

Secondary Task 2
Secondary Task 1
Primary Task

High End of Memory
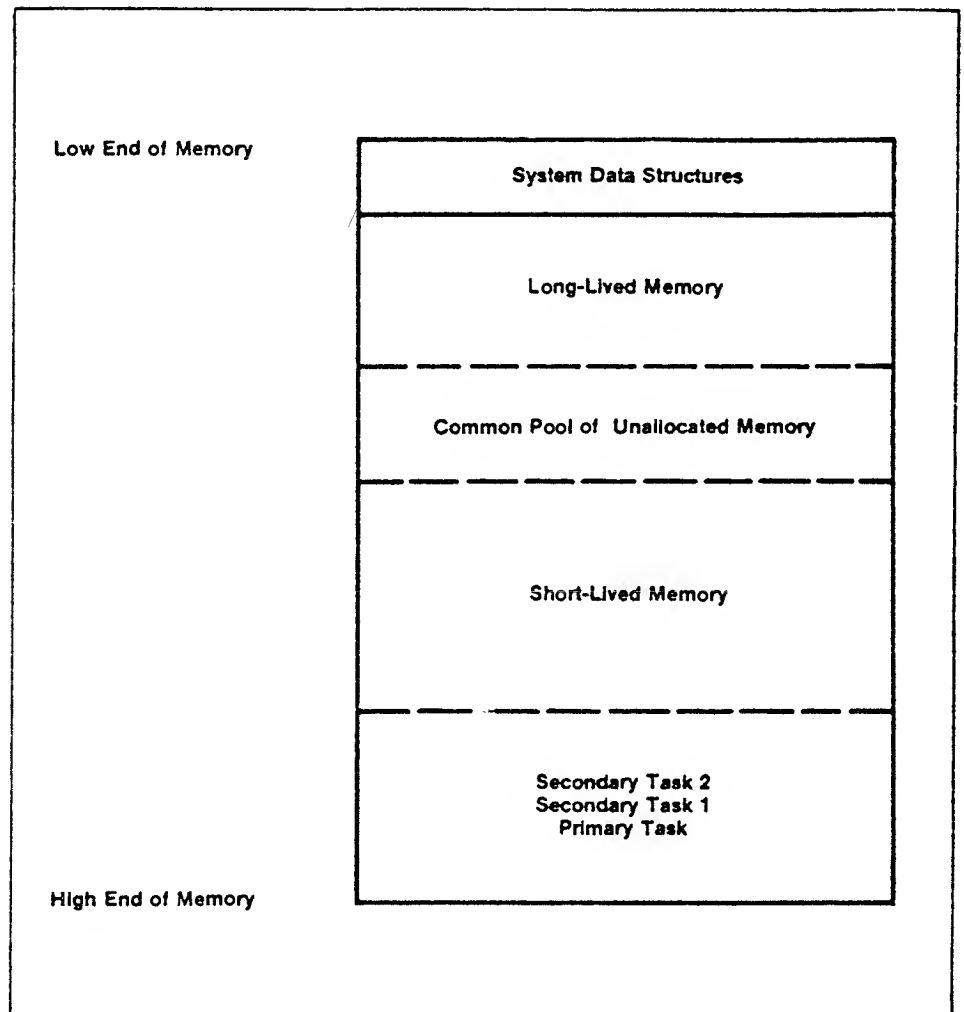
Figure 10-3. Memory Organization of an Application Partition

10-5

Creating Secondary Application Partitions

Secondary application partitions can be created and loaded either at system initialization or dynamically during execution.

At System Initialization

A user can create and load secondary application partitions through a batch job control file that is processed during system initialization. System services such as the printer spooler, ISAM, or RJE can be loaded in this way.

Dynamically

A process residing in the primary application partition can create secondary application partitions dynamically with the CreatePartition operation. Each new partition is created from the high-address end of the primary application partition. The remaining memory is redefined as the primary application partition. CreatePartition specifies a partition name, returns a partition handle, and causes the exit run file (see below) to be loaded immediately into the primary application partition, replacing the application system that executed the operation.

Partition Handle

A partition handle is a 16-bit integer that uniquely identifies a secondary application partition. It is returned by the CreatePartition operation and is used to refer to the partition in subsequent operations such as GetPartitionStatus, LoadPrimaryTask, and RemovePartition.

A process can obtain a previously assigned partition handle by supplying the partition name when using a GetPartitionHandle operation.

Loading Tasks

A secondary application partition is vacant when created. A process in the primary application partition loads and activates the first task, called the primary task, in a secondary application partition with the LoadPrimaryTask operation.

The primary task in turn can load additional tasks, called secondary tasks, in its own partition with the LoadTask operation. (See the "Task Management" section.)

Exit Run File

An exit run file is a user-specified file that is loaded and activated when the executing application system exits. Each application partition has its own exit run file. (See the "Task Management" section.)

In the primary application partition, if no exit run file is specified, the system will malfunction and reboot itself. If the exit run file cannot be read, it displays the message "Cannot load exit run file" and a status code indicating the type of error that occurred. If the exit run file is on a floppy disk, the user can insert a floppy disk with the appropriate exit run file and the system will resume loading the exit run file.

In a secondary application partition, if no exit run file is specified or if it cannot be read, the partition becomes vacant.

Obtaining Partition Status

A process can obtain status information about a specified application partition and the job executing in it with the GetPartitionStatus operation. The process can obtain any of the following: User Control Block, Partition Descriptor, or Batch Control Block. (See "System Data Structures" below.)

Interpartition Communication

A process in one application partition can send messages to a process in another application partition. The destination process first allocates an exchange and makes the exchange known to the OS with the SetPartitionExchange operation. The sender process obtains the exchange number with the GetPartitionExchange operation, then sends messages to the exchange.

The processes engaged in the interpartition communications must lock themselves into their respective partitions with the SetPartitionLock operation to avoid being terminated by a TerminatePartitionTasks or VacatePartition operation from the primary application partition. The termination of an application system that is currently engaged in interpartition communication will result in unpredictable system malfunction.

Terminating Tasks

A process terminates the entire application system in its own partition by using the Chain, Exit, or ErrorExit operation. (See the "Task Management" section.)

In addition, two operations can be executed in the primary application partition to terminate the application system in a specified secondary application partition:

TerminatePartitionTasks terminates all tasks in the specified secondary application partition and loads and activates the partition's exit run file, if one is specified.

VacatePartition terminates all tasks in the specified secondary application partition but does not load and activate the partition's exit run file. VacatePartition leaves the partition vacant.

Removing Partitions

A process in the primary application partition can remove an existing secondary application partition that is vacant with the RemovePartition operation.

A secondary application partition is vacant when:

it is first created,

the current application system exits with no exit run file specified, or

the VacatePartition operation is performed.

10-8

If a secondary application partition adjacent to the primary application partition is removed, the memory it occupied becomes part of the primary application partition.

If a secondary application partition that is not adjacent to the primary partition is removed, the memory it occupied becomes a block of unused memory. Adjacent blocks of unused memory are combined into a single block. Such blocks serve as a pool of unallocated memory from which new application partitions are created using a first-fit algorithm.

Deallocation of System Resources

In a compact system, all allocated resources are deallocated when the application system exits. (Examples of allocated resources are exchanges, file handles, and timer requests.)

In a system where multiple application systems can be executed simultaneously, only the resources allocated to an exiting application system are deallocated. Information on the resource allocations of each application system is stored in application partition data structures that augment but do not replace the data structures present in the compact configuration.

Application Partition Data Structures

The application partition management facility maintains six data structures for each application partition. These data structures are described in the order shown in Figure 10-4 (from the left side, top to bottom):

Extended User Control Block (Extended UCB), which contains the offset of the Partition Descriptor.

Partition Descriptor, which contains the partition name, and the boundaries of the partition and of its long- and short-lived memory areas. It also contains internal links to partition descriptors in other partitions.
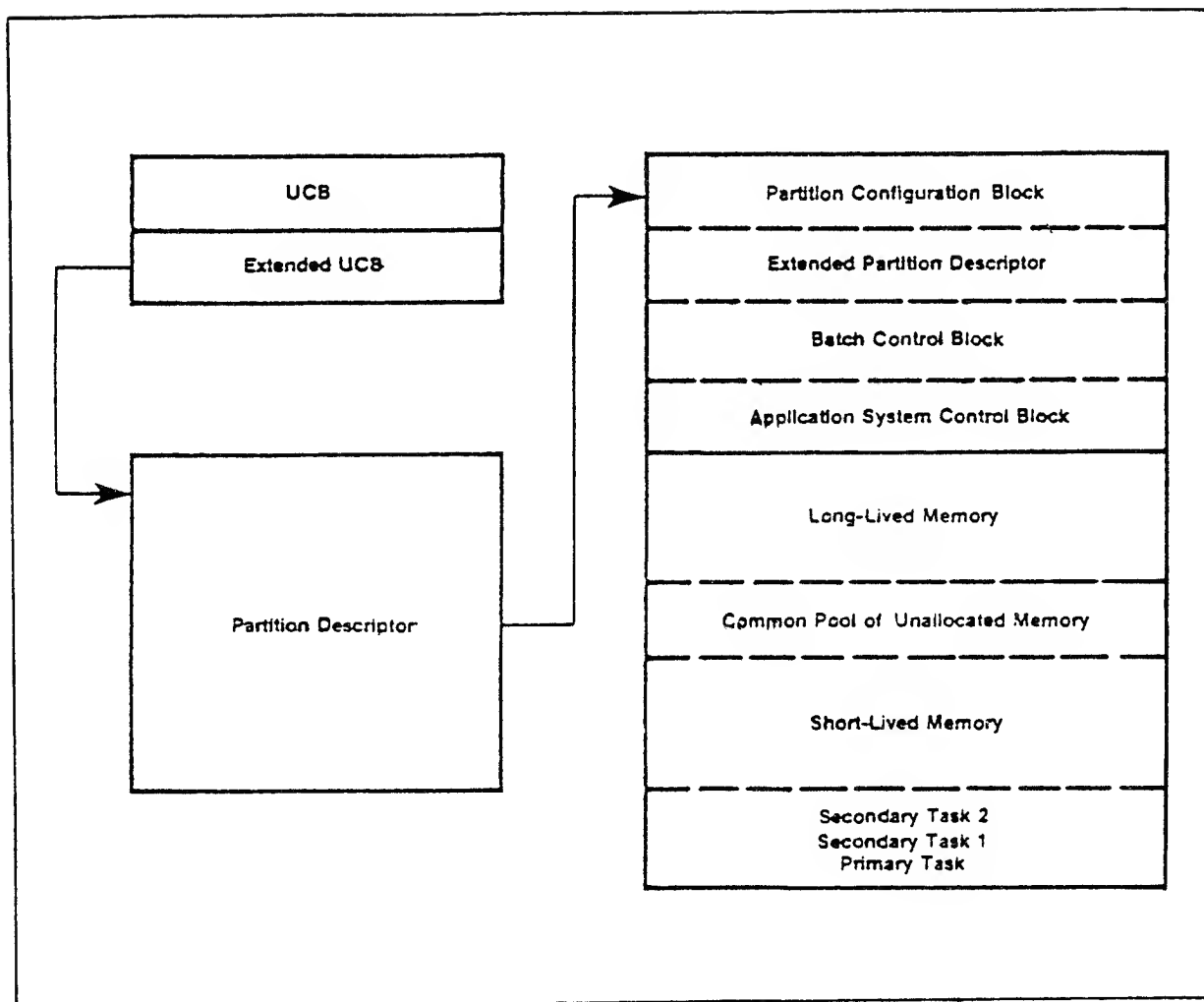
Figure 10-4. Application Partition Data Structures

Partition Configuration Block, which contains the offsets of the Extended Partition Descriptor, Batch Control Block, and Application System Control Block.

Extended Partition Descriptor, which contains specifications for the current application system and exit run file.

Batch Control Block, which contains the job name and class, file handle and logical file address of the batch job control file, Assigned Device Block, and SysIn and SysOut byte stream work area and buffers. This data structure is used by the batch manager.

Application System Control Block (see the "Parameter Management" section), which communicates parameters between application systems.

The format for each data structure is given in Appendix E.

OPERATIONS: SERVICES

Application partition management operations are categorized by function in Table 10-1 below.

Table 10-1. Application Partition Management Operations by Function

---

Interpartition Communication	Partition Control
	CreatePartition
GetPartitionExchange	GetPartitionHandle
SetPartitionExchange	GetPartitionStatus
SetPartitionLock	RemovePartition

Task Control

LoadPrimaryTask
TerminatePartitionTasks
VacatePartition

Interpartition Communication

GetPartitionExchange
> gets the exchange number set up by the SetPartitionExchange operation.

SetPartitionExchange
> sets up an exchange number that can be queried by a task in another application partition for interpartition communication.

SetPartitionLock
> declares whether an application system in the specified application partition can be terminated by the TerminatePartitionTasks or VacatePartition operation.

Partition Control

CreatePartition
>   creates a new secondary application
>   partition, assigns its name, and
>   returns a partition handle.

GetPartitionHandle
>   translates the name of the specified
>   application partition into a
>   partition handle.

GetPartitionStatus
>   returns status information about the
>   specified application partition and
>   the job currently executing in it.

RemovePartition
>   removes the specified vacant
>   application partition.

Task Control

LoadPrimaryTask
>   loads and activates a primary task
>   run file in the vacant application
>   partition specified by the partition
>   handle.

TerminatePartitionTasks
>   terminates all tasks in the
>   application partition specified by
>   the partition handle and and loads
>   the partition's exit run file.

VacatePartition
>   terminates all tasks in the
>   application partition specified by
>   the partition handle but does not
>   load the partition's exit run file.
>   VacatePartition leaves the partition
>   vacant.

CreatePartition

Description

The CreatePartition service creates a new application partition, assigns its name, and returns a partition handle. CreatePartition can be issued only by a process executing in the primary application partition.

CreatePartition causes the exit run file to be loaded into the primary application partition, replacing the application system that executed the CreatePartition operation.

Procedural Interface

CreatePartition (pbPartitionName, cbPartitionName, cParagraph, fRunBatch, pPhRet): ErcType

where

pbPartitionName
cbPartitionName

describe the partition name (up to 12 characters).

cParagraph   is the number of paragraphs of memory to be allocated to the application partition.

fRunBatch   is TRUE or FALSE. TRUE indicates a Batch Control Block of 1.3 kilobytes is allocated in addition to the memory for the partition itself. FALSE indicates no Batch Control Block is allocated.

pPhRet   is the memory address of the word into which the partition handle is returned.

Request Block

sPhRet is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	181
12	cParagraph	2	
14	fRunBatch	1	
15	reserved	3	
18	pbPartitionName	4	
22	cbPartitionName	2	
24	pPhRet	4	
28	sPhRet	2	2

GetPartitionExchange

Description

The GetPartitionExchange service returns the exchange number established by the SetPartitionExchange operation. The exchange number is used to communicate with an application system executing in another application partition.

Procedural Interface

GetPartitionExchange (ph, pExchRet): ErcType

where

ph   is the partition handle returned from a CreatePartition or GetPartitionHandle operation. A 0 specifies the application partition in which the client process is executing.

pExchRet   is a pointer to a 16-bit word into which the exchange is returned.

Request Block

sExchRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	UserNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	184
12	ph	2	
14	pExchRet	4	
16	sExchRet	2	2

GetPartitionHandle

Description

> The GetPartitionHandle service translates the specified application partition name into a partition handle.

Procedural Interface

> GetPartitionHandle (pbPartitionName,
>   cbPartitionName,
>   pPhRet): ErcType
>
> where
>
> pbPartitionName
>
> cbPartitionName
>
> > describe the partition name (up to 12 characters).
>
> pPhRet   is the memory address of the word into which the partition handle is returned.

Request Block

> sPhRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	177
12	pbPartitionName	4	
16	cbPartitionName	2	
18	pPhRet	4	
22	sPhRet	2	2

GetPartitionStatus

Description

The GetPartitionStatus service returns status
information about the specified application
partition and the job currently executing in it.

Procedural Interface

GetPartitionStatus (ph, statusCode, pStatusRet,
                    sStatusMax): ErcType

where

ph   is the partition handle returned
                from a CreatePartition or
                GetPartitionHandle operation. A 0
                specifies the application partition
                in which the client process is
                executing.

statusCode   specifies the status code. The
                status items and values are:

Code	Item	Size
0	Partition Descriptor	33
1	Extended Partition	
	Descriptor	172
2	Batch Control Block	1548
3	Application System	
	Control Block	280

pStatusRet
sStatusMax   describe the memory area to which
                the status information is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	185
12	ph	2	
14	statusCode	2	
16	pStatusRet	4	
20	sStatusMax	2	

LoadPrimaryTask

Description

The LoadPrimaryTask service loads and activates the primary task run file in the vacant application partition specified by the file specification.

LoadPrimaryTask:

1. Verifies that the file specification specifies a run file that contains a valid task image and that the task image fits in the application partition.

2. Allocates a short-lived memory segment large enough to contain the task image from the specified run file.

3. Reads the task image from the run file into the application partition.

4. Relocates all intersegment references to accommodate the memory address at which the task image is loaded.

5. Creates a process to be scheduled at the specified priority. The initial values loaded into the segment registers (CS, DS, SS, ES), the Stack Pointer (SP), and the Instruction Pointer (IP) are derived from information in the run-file header. (See the "Task Management" section.)

Procedural Interface

```
LoadPrimaryTask   (ph, pbFileSpec, cbFileSpec,
                   pbPassWord, cbPassWord,
                   priority):  ErcType
```

where

ph   is the partition handle returned from a CreatePartition or GetPartitionHandle operation.

pbFileSpec
cbFileSpec   describes a character string of the form {node}[volname]<dirname>file-name. The distinction between uppercase and lowercase in file specifications is not significant in matching file names.

pbPassWord
cbPassWord   describes the volume, directory, or
              file password that authorizes
              access to the specified file.

priority   is the priority (0-254, with 0 the
              highest) at which to schedule the
              newly   created   process   for
              execution.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	178
12	ph	2	
14	priority	2	
16	reserved	2	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassWord	4	
28	cbPassWord	2	

RemovePartition

Description

The RemovePartition service removes the specified vacant application partition. RemovePartition can be issued only from a process executing in the primary application partition.

Procedural Interface

RemovePartition (ph): ErcType

where

ph   is the partition handle returned
                      from a CreatePartition or
                      GetPartitionHandle operation. A 0
                      specifies the application partition
                      in which the client process is
                      executing.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	176
12	ph	2	

SetPartitionExchange

Description

> The SetPartitionExchange service sets up an exchange number that can be queried by a task in another application partition for interpartition communication. The application system should use the SetPartitionLock operation before using SetPartitionExchange to ensure the integrity of its operation.

Procedural Interface

> SetPartitionExchange (exchange): ErcType
>
> exchange   is an exchange previously allocated by the application system.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	183
12	exchange	2	

SetPartitionLock

Description

The SetPartitionLock service declares whether an
application system executing in the specified
application partition is locked. If it is
locked, it cannot be terminated by the
TerminatePartitionTasks or VacatePartition
operation. An application system can lock itself
into its own partition only.

An Exit or ErrorExit from an application system
in the locked partition vacates the application
partition, but no other run file is loaded and
the partition cannot be deleted except by system
reload.

Procedural Interface

SetPartitionLock: (fLock) ErcType

where

fLock	is TRUE or FALSE. TRUE means that the partition is locked. FALSE means that the partition is unlocked.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	182
12	fLock	1	
13	reserved		

TerminatePartitionTasks

Description

The TerminatePartitionTasks service terminates
all tasks in the application partition specified
by the partition handle and loads and activates
the partition's exit run file.

If the partition is locked, a status code is
returned and a flag is set in the Application
System Control Block to notify the task in the
partition.

Procedural Interface

TerminatePartitionTasks (ph): ErcType

where

ph   is the partition handle returned
                from a CreatePartition or
                GetPartitionHandle operation. A 0
                specifies the application partition
                in which the client process is
                executing.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	179
12	ph	2	

VacatePartition

Description

>The VacatePartition service terminates all tasks in the application partition specified by the partition handle but does not load and activate the exit run file. VacatePartition leaves the partition vacant.
>
>If the partition is locked, a status code is returned and a flag is set in the Application System Control Block to notify the task in the partition.

Procedural Interface

>VacatePartition (ph): ErcType
>
>where
>
>ph   is the partition handle returned from a CreatePartition or GetPartitionHandle operation. A 0 specifies the application partition in which the client process is executing.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	180
12	ph	2	

SECTION 11
CLUSTER MANAGEMENT

OVERVIEW

One high-speed RS-422 channel is standard on each workstation. In small cluster configurations (up to four cluster workstations), the master workstation uses this Channel for communications with the cluster workstations. However, in large cluster configurations, the master B22 workstation uses one or two Communications I/O Processors (CommIOPs) for communications with the cluster workstations.

The CommIOP, which is added to the Multibus of the B22, is an intelligent communications processor based on the Intel 8085 microprocessor. The CommIOP serves up to four cluster workstations on each of its two high-speed serial input/output channels. (The CommIOP can actually handle up to 15 cluster workstations per high-speed line if they have their own local file system and only occasionally access files on the master workstation.)

CommIOP software consists of an 8085 bootstrap-ROM program, the main CommIOP program (which executes in 8085 RAM), and a CommIOP handler (written in 8086 code) which executes in system memory under OS control.

CONCEPTS

The CommIOP is an intelligent communications processor based on the Intel 8085 microprocessor. The CommIOP serves up to four cluster workstations on each of its two high-speed serial input/output (SIO) channels.

Software

CommIOP software consists of:

o   the 8085 bootstrap-ROM program (used for self-tests),

o   the main CommIOP program (which executes in 8085 RAM), and

o   the CommIOP handler (written in 8086 code) which executes in system memory under OS control.

Initialization

The CommIOP and the master workstation communicate using interprocessor interrupts and shared memory. When the master workstation is turned on or its reset button pushed, the CommIOP performs a self-test using the 8085 bootstrap-ROM program and waits for an interrupt from the 8086 processor.

The CommIOP handler initializes each CommIOP in four steps. (The number of CommIOPs is a system build parameter.) The CommIOP:

1.   acknowledges to the CommIOP handler that it is functioning,

2.   runs a memory test in its RAM,

3.   loads the main CommIOP program into its RAM from system memory, and

4.   starts operation.

The CommIOP and the CommIOP handler communicate using an initialization control block located in

system memory at locations 01E0h-01EFh. The CommIOP acknowledges completion of each of the above steps by writing a completion status in the initialization control block.

The CommIOP can also, as part of its initialization, (1) dump the contents of its RAM into system memory (this is useful for debugging), and (2) test system memory. These two functions are system build parameters and occur, if requested, after steps 1 and 2 above, respectively.

Operation

Before the main CommIOP program actually starts operation, the CommIOP handler establishes queues in system memory for its use. These queues contain addresses of buffers used by the CommIOP to copy requests from each cluster workstation.

As a request comes in from the cluster workstation, the main CommIOP program obtains a buffer, copies the request into it, and places the request on the inbound request queue. The CommIOP handler removes the request from the inbound request queue and submits it to the master workstation Agent Service Process, which then submits it to the appropriate system service process.

After the request is processed, it is returned to the CommIOP handler, which places it on the outbound data queue. The CommIOP then copies the request into its own RAM and returns the response to the appropriate cluster workstation.

The CommIOP interrupts the master workstation only when it deposits data onto a previously empty queue. The Communications Interrupt Service Routine in the processor sends a message to the exchange of the CommIOP handler to awaken it.

The maximum number of requests a cluster workstation can have outstanding is three.

Status

At regular intervals, the CommIOP updates a status block in system memory. The CommIOP inserts a status code into this block if it detects any irrecoverable errors (hardware malfunction, invalid control structures, etc.). The content of the status block is returned by the GetClusterStatus operation.

The master workstation (with or without a CommIOP) keeps statistics about errors and normal operational parameters. The GetClusterStatus operation makes these statistics available to any workstation.

OPERATIONS: SERVICES

Cluster management provides the operations listed below.

DisableCluster   allows an application system on
                     the master workstation to
                     disable polling of the cluster
                     workstations after a specified
                     time period. DisableCluster
                     is also used to resume polling
                     of the cluster workstations.

GetClusterStatus   returns usage statistics for
                     each communications channel and
                     the workstations attached to
                     it.

GetWSUserName   returns the user name that is
                     signed on at the specified
                     workstation.

SetWSUserName   stores the user signon name of
                     the workstation.

DisableCluster

Description

The DisableCluster service allows an application system on the master workstation to disable polling of the cluster workstations after a specified time period. DisableCluster is also used to resume polling of the cluster workstations.

During the specified time period, the GetDateTime operation (see the "Timer Management" section) returns the "Master workstation going down" status code and the time left (in seconds) before polling stops. After the specified time period, all operations return status code 46 ("Master workstation going down").

Typically, the application system (for example, the Executive in the primary application partition) on the cluster workstation that performs the GetDateTime operation would notify the cluster workstation user when it received the "Master workstation going down" status code.

DisableCluster is useful for stopping all cluster workstation activity, for example, to perform software maintenance on the master workstation.

Procedural Interface

DisableCluster (fDisablePoll,
                timeInterval): ErcType

where

fDisablePoll
        disables polling if TRUE or resumes it if FALSE.

timeInterval
        is the time period (in tenths of a second). This is not meaningful if fDisablePoll is FALSE.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	122
12	fDisablePoll	2	
14	timeInterval	2	

GetClusterStatus

Description

The GetClusterStatus service returns usage statistics for each communications line and the workstations attached to it.

The communications channels are identified as follows:

Channel Number	Communications Channel
0	standard channel
1	CommIOP 1, Channel A
2	CommIOP 1, Channel B
3	CommIOP 2, Channel A
4	CommIOP 2, Channel B

Procedural Interface

GetClusterStatus (iLine, pBufferRet,
                  sBufferMax): ErcType

where

iLine   is the communications channel number.

pBufferRet is the memory address of the buffer into which to place the communications status buffer (see Table 11-1, below).

sBufferMax is the size of the buffer. If the buffer is too small, the statistics are truncated.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	100
12	iLine	2	
14	reserved	4	
18	pBufferRet	4	
22	sBufferMax	2	

Table 11-1. Communications Status Buffer

Offset	Field	Size (bytes)
0	nWsConf	1
1	nWsActive	1
2	Up	4
6	Idle	4
10	Ops	4
14	ErrTO	4
18	ErrCRC	4
22	ErrOvrn	4
26	ErrGen	4
30	rgWsStatus	n*16

where

nWsConf   is the number of workstations configured for this communications channel at system build.

nWsActive   is the number of workstations currently active.

Up	are the number of 100 ms intervals elapsed since the communications channel was activated.
Idle	are the number of 100 ms intervals elapsed in which the communications channel was inactive.
Ops	are the number of operations performed on this communications channel.
ErrTO	is the number of time out errors.
ErrCRC	is the number of cyclic redundancy check errors.
ErrOvrn	is the number of overrun errors.
ErrGen	is the number of sequence and other nonclassified errors.
rgWsStatus	is an array of $\underline{n}$ workstation status blocks, where $\underline{n}$ is the number of configured workstations. The format of each block is shown in Table 11-2 below.

Table 11-2. wsStatus Block

Offset	Field	Size (bytes)
0	iUserNum	1
1	fActive	1
2	RxRq	4
6	nRqs	2
8	reserved	8

where

iUserNum	is the user number associated with the workstation.
fActive	is the workstation active flag. The workstation is inactive if it is 0, and active if it is 0FFh.
RxRq	is the number of requests received during the current session for the workstation.
nRqs	is the number of pending requests for this workstation.

GetWSUserName

Description

> The GetWSUserName service returns the user name
> that is signed on at the specified workstation
> (used with the GetClusterStatus operation to find
> the user names of active workstations). (See the
> description of the GetClusterStatus operation.)

Procedural Interface

> GetWSUserName (WSNum, pWSUserNameRet,
>   sWSUserNameRetMax): ErcType

> where

> WSNum   is the workstation identification
>   number.

> pWSUserNameRet
>   is the memory address of the first
>   byte of the buffer to which the user
>   name is returned. The first byte of
>   the character string is the size
>   (maximum of 31).

> sWSUserNameRetMax
>   is the size of the buffer.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	202
12	WSNum	2	
14	reserved	4	
18	pWSUserNameRet	4	
22	sWSUserNameRetMax	2	

SetWSUserName

Description

The SetWSUserName service stores the user signon name of the workstation. SetWSUserName is used primarily by the Signon program which also places the user name in the Application System Control Block of the master workstation in a cluster configuration.
(See the "Application System Control Block" subsection in the "Parameter Management" section.)

Procedural Interface

SetWSUserName (pbUserName, cbUserName): ErcType

where

pbUsername
cbUserName   describe the user signon name to be stored.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	203
12	reserved	6	
18	pbUserName	4	
22	cbUserName	2	

SECTION 12
NETWORK MANAGEMENT

(To be supplied)

.

SECTION 13
SYSTEM SERVICES MANAGEMENT

OVERVIEW

The Operating System includes a number of system service processes. These processes, which are scheduled for execution in the same manner as application processes, receive IPC messages to request the performance of their services. Any process, even a system service process, can use (be a client of) a system service process.

Each system service process acts as the system-wide guardian and manager for a class of system resources, such as files, memory, or keyboard. Because the system service process is the only software element that accesses the resource, and because the interface to the system service process is formalized through the use of IPC, a highly modular environment results.

This modular environment increases reliability by localizing the scope of processing and provides the flexibility to replace a system service process as a complete entity.

System builders can also include their own system service processes, which are then indistinguishable from Burroughs processes.

In the "Interprocess Communication Management" section, see the following subsections for more details: "System Service Processes," "Accessing System Services," "Procedural Access to System Services," "Direct Access to System Services," "Interaction of Client Processes and System Service Processes," "Filter Processes," "Request Blocks," "Request Primitive", "Respond Primitive," "Wait Primitive," and "Interstation Communication."

CONCEPTS

A system service process can be added to the Operating System in three ways. It can be:

linking into the OS System Image,

dynamically installed in an extended system partition, or

dynamically installed in a secondary application partition.

The request codes served by a dynamically installed system service process must be reserved at system build. (See the B20 System Programmers and Assembler Reference Manual (Part 1), form 1148699 for details on linking into the OS System Image and on system build.)

Dynamically Installing a System Service in an Extended System Partition

A system service process that is to be dynamically installed in an extended system partition is first linked into its own self-contained task image. The system service process must be self-installing. It is installed with the Executive's Run File command or its own command (created with the Executive's New Command command). (See the B20 System Executive Reference Manual, form 1144474 for more information about these two commands.)

Once installed, the system service is permanent and cannot be removed except by system reload. The location of the extended system partition depends on whether any secondary application partitions have been created before the system service is installed.

If the system service process is installed before any secondary application partition is created, the extended system partition is placed at the highest available memory location.

If the system service process is installed after a secondary application partition is created, the extended system partition is located between the secondary and primary application partitions.

A typical sequence of operations for a self-installing system service process might include:

1. The ChangePriority operation (see the "Process Management" section) to change its priority appropriately.

2. The AllocMemorySL or DeallocMemorySL operation to allocate/deallocate its short-lived memory segment (if different from the size of its run file). It may be efficient to place initialization code (which is never reused) in the lowest address locations of the task image and to use DeallocMemorySL to deallocate the memory that contains this initialization code before step 6 below.

3. The AllocExch operation (see the "Exchange Management" section) to allocate a service exchange and any other exchanges needed for its internal operation.

4. The CreateProcess operation (see the "Process Management" section) to create any additional processes needed for its operation.

5. The ServeRq operation (described later in this Section) for each request code it is to serve. Specify the service exchange allocated in step 3 above. The number of request codes is specified at system build.

   Request codes 0 through 7FFFh are reserved for future expansion and should not be used by system builders. Request codes 8000h-0FFFFh are available for system builder use. (Appendix D lists the request codes.)

6. The ConvertToSys operation (described later in this Section) to convert its processes, short-lived memory, and exchanges to system service processes, system memory, and system exchanges, respectively. This prevents these resources from being released during Chain, ErrorExit, and Exit operations (see the "Task Management" section).

7. The Chain operation to load the specified exit run file as the succeeding application system. Chain normally returns to the application system only if it fails.

13-3

However, in the special case described in this sequence, Chain always returns. Therefore the call to Chain should not be in the initialization code that is overlaid by the new application system.

Restrictions

The following restrictions are to avoid conflict with the Chain, ErrorExit, and Exit operations initiated by the application system. After using ConvertToSys in step 6 above, a system service process <u>must</u>:

o   use the OpenFileLL, rather than the OpenFile, operation (described in the "File Management" section),

o   not use the CloseAllFiles or CloseAllFilesLL operations (described in the "File Management" section),

o   not allocate or deallocate exchanges,

o   not allocate or deallocate memory,

o   not create processes, and

o   not use the Chain (other than as described in step 7 above), ErrorExit, or Exit operations.

Dynamically Installing a System Service in a Secondary Application Partition

A system service is installed in a secondary application partition by the process executing in the primary application partition with the LoadPrimaryTask operation. (See the "Application Partition Management" section.) A system service installed in a secondary partition is not permanently installed and can be removed by application partition management operations.

A system service executing in a secondary application partition must not use the ConvertToSys operation, because doing so would prevent its dynamic removal. Also, it should use the Chain operation only to remove itself, not to replace the current application system with a specified run file. Since a system service in a secondary application partition has a unique user number, it is not subject to the restrictions noted above for a system service installed in an extended system partition.

13-4

OPERATIONS: SERVICES

System services management provides the operations listed below.

ConvertToSys   converts all processes, short-lived memory, and exchanges in the primary application partition to system service processes, system memory, and system exchanges, respectively.

ServeRq   used by a dynamically installed system service process to declare its readiness to serve the specified request code.

ConvertToSys

Description

The ConvertToSys service converts all processes, short-lived memory, and exchanges in the primary application partition to system service processes, system memory, and system exchanges, respectively.

Procedural Interface

ConvertToSys: ErcType

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	98

ServeRq

Description

The ServRq service is used by a system service process that is dynamically installed to serve the specified request code. Future requests containing the specified request code are queued at the specified exchange.

Specifying exchange 0 indicates that the calling process is no longer serving the specified request code. However, this does not dequeue currently queued requests at the exchange that was formerly associated with the specified request code. Status code 33 ("Service not available") is returned to future requests containing the specified request code.

Procedural Interface

ServeRq (requestCode, exchange): ErcType

where

requestCode   is the request code.

exchange   is the service exchange number or 0.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	99
12	requestCode	2	
14	exchange	2	

SECTION 14

FILE MANAGEMENT

OVERVIEW

The file management system provides a hierarchical organization of disk file data by node, volume, directory, and file. Volumes are automatically recognized when placed online. Each file can have a 50-character file name, a 12-character password, and a file protection level. A file can be dynamically expanded and contracted without limit as long as it fits on one disk. Concurrent access is controlled by read (shared) and modify (exclusive) access modes.

While providing convenience and reliability, the file management system supplies the system builder with the full throughput capability of the disk hardware. This includes reading or writing any 512-byte sector of an open file with one disk access, reading or writing up to 65K bytes with one disk operation, input/output overlapped with process execution, and optimized disk arm scheduling.

In a cluster configuration, files can be located at cluster workstations as well as the master workstation.

The file management services of the OS are efficient, reliable, and convenient to use.

Efficiency is provided through:

careful data placement.

The volume control structures resident on each volume are placed to minimize disk arm movement.

The Volume Home Block is brought into memory when a volume is placed online. In addition, the most recently used directory information is retained in memory.

randomization (hashing) techniques.

These techniques reduce the number of disk reads required to access directory information. These techniques are used for placing an entry in a directory and are later used for locating it.

Reliability is provided through:

o   multilevel (volume, directory, or file) password protection.

o   multiple file protection levels.

   A file protection level specifies the access allowed to a file when the acessing process does not present a valid volume or directory password.

o   duplication of two volume control structures: the Volume Home Block and the File Header Blocks.

   This duplication ensures that damage to one copy of a volume control structure does not cause a loss of data.

Convenience is provided through:

o   hierarchical organization of disk file data by node, volume, directory, and file.

o   long file names (up to 50 characters).

o   dynamic file length.

   The user determines the file length when the file is created and can change it later.

o   removable file volumes (floppy disks).

o   automatic recognition of volumes placed online.

o   read (shared) or modify (exclusive) file modes.

o   device independence.

   The device a file is located on is transparent to the user.

File Access Methods

File access methods augment the file management system by providing more structured access to disk file data. There are four file access methods:

o   the Sequential Access Method,

o   the Record Sequential Access Method,

o   the Direct Access Method, and

o   the Indexed Sequential Access Method (ISAM).

The first three access methods are described in detail in this manual. The fourth access method is described in the B20 System ISAM Reference Manual, form 1148723.

Local File System

A cluster workstation can have its own local file system. The local file system allows a cluster workstation to access files on local mass storage as well as files on mass storage at the master workstation. The file system filter process of the cluster workstation intercepts each file access request and directs it either to the local file system or to the master workstation.

A cluster workstation can be bootstrapped either from a file at the master workstation or from its local file system. A cluster workstation bootstrapped from its local file system is a self-contained entity that accesses the master workstation only for shared files. If a malfunction occurs at the master workstation, the cluster workstation can continue to operate normally, provided all file accesses are to local mass storage.

CONCEPTS

The file management system has a hierarchical organization of disk file data by node, volume, directory, and file.

Node

A B20 system connected to a standard network can access the files of other network nodes, subject to password protection. The node at which a file is located must be specified for files not located at the same node as the requesting process.

Volume

The files of the system are located on volumes. A volume is the media of a disk drive that was formatted and initialized using the IVolume utility. It is protected by a volume password. (See the B20 System Software Operation Guide, form 1148772.)

For example, a floppy disk and the media sealed inside a Winchester disk drive are volumes and a floppy disk is a removable volume.

A volume contains a number of volume control structures: the Volume Home Block, the File Header Blocks, and the Master File Directory, among others. (These structures are described in detail in "Volume Control Structures" below.)

The Volume Home Block is the root structure of information of a disk volume. The File Header Block of each file contains information about that file and about the disk address and size of each of its Disk Extents. (A Disk Extent is one or more contiguous disk sectors.) The Master File Directory (which contains an entry for each directory on the volume) and the directories provide fast access to the File Header Block of a specific file. They do not, however, contain any information about the file that is not also contained in its File Header Block.

There are duplicate Volume Home Blocks (working and initial copies) and duplicate File Header Blocks (primary and secondary copies) on the volume for reliability. The primary and secondary copies of the File Header Blocks are

14-4

located on different cylinders and at different rotational positions and are accessed (except for floppy disks) by different read/write heads. These duplicates ensure that damage to one copy does not cause a loss of data.

The location on the volume of the Volume Home Blocks, the File Header Blocks, and the other volume control structures minimizes disk arm movement and therefore maximizes efficiency. The File Header Blocks are located in a single area of the volume, the disk address and size of which are recorded in the working and initial copies of the Volume Home Block. Volume control structures that are frequently accessed, including the primary and secondary copies of the File Header Blocks, are located near the middle of the disk.

Directory

The files of a volume are divided into one or more directories. A directory is a collection of related files on one volume. The maximum number of directories that can be created on a volume depends on the size of the Master File Directory; its size is specified when the volume is initialized. The maximum number of files that can be created in a directory depends on the directory's size, which is specified when the directory is created. A directory is protected by a directory password.

A directory is created with the CreateDir operation and deleted with the DeleteDir operation.

File

A <u>file</u> is a set of related records (on disk) treated as a unit. The files of a volume consist of integral numbers of 512-byte sectors and must be completely contained on it. There are no other restrictions on file size. A file is protected by a file protection level and by an optional file password.

A file is created with the CreateFile operation and deleted with the DeleteFile operation. Once it is created, it is accessed with the OpenFile operation and closed with the CloseFile operation. The ChangeFileLength operation

changes the length of an open file. The RenameFile operation renames an existing file.

Automatic Volume Recognition

The OS automatically recognizes volumes placed online (that is, mounted). For example, when a floppy disk is inserted into a disk drive, the OS reads the disk to determine whether it contains a volume and, if it does, that no other volume of the same name is already online. After this validation by the OS, the volume responds to user requests containing appropriate specifications and passwords.

When a volume is placed online, the Volume Home Block is read into memory. It remains there as long as the volume remains online.

If a floppy drive door is opened, any open files on the disk in that drive are automatically put into a special dismounted state. Such files can be closed as usual, but any attempt to perform other operations on them returns status code 216 ("Wrong volume mounted").

Node Name

A node name is a string of characters. It can have a maximum of 12 characters.

Volume Name

A volname (volume name) is a string of characters. It can have a maximum of 12 characters.

System Volume

The volume on which the OS resides can be referenced in two ways: by its synonym, Sys, or by the name it was given when it was initialized with the IVolume utility.

In a master or standalone workstation, Sys is a synonym for the volume name of the device from which the OS is bootstrapped.

For example, in a dual-floppy standalone system, where the OS is bootstrapped from the floppy disk in drive 0, Sys can be used instead of the volume name of the floppy disk in drive 0. In a Winchester-based (hard disk) system, where the OS is bootstrapped from hard-disk drive 0, Sys can be used instead of its volume name.

In a cluster workstation without local disk storage, Sys is a synonym for the volume name of the device from which the master workstation of that cluster system is bootstrapped.

In a cluster workstation bootstrapped from its local disk, Sys is a synonym for the volume name of the device from which the cluster workstation is bootstrapped.

!Sys is a synonym for the volume name of the device from which the master workstation of the cluster is bootstrapped.

Scratch Volume

The volume on which scratch (temporary) files are placed can be referenced either by its synonym, Scr, or by its real name.

Directory Name

A <u>dirname</u> (directory name) is a string of characters. It can have a maximum of 12 characters.

File Name

A <u>filename</u> (file name) is a string of characters. It can have a maximum of 50 characters.

Directory and File Specifications

A directory is referred to with a directory specification. A <u>directory specification</u> has the form:

{node}[volname]dirname

A file is referred to with a file specification. A <u>full file specification</u> has the form:

{node} [volname]<dirname>filename

The distinction between uppercase and lowercase in directory and file specifications is not significant in matching directory and/or file names during directory lookup; the distinction is, however, preserved by the file management system to make the directory and file specifications easier to read. For example, a file can be created with the specification:

[MasterVol]<Susan>Todays>work

The same file can later be accessed as:

[mastervol]<SUSAN>todays>Work

It is recommended that node names, volume names, and directory names consist only of alphanumeric characters, plus the period, ".", and the hyphen, "-". It is recommended that file names consist of alphanumeric characters, plus the period, ".", the hyphen, "-", and the right angle bracket, ">".

Abbreviated Specifications

A file or directory can be referred to with an abbreviated specification if default specifications were previously established.

The SetPath operation establishes a default node, a default volume, a default directory, and a default password. (See "Passwords" below.) The SetPrefix operation establishes a default file prefix. SetPath and SetPrefix establish defaults for the user number of the calling process. A unique user number is associated with each application partition.

If a SetPath operation is issued with the default volname of [MasterVol] and the default dirname of <Susan>, the user can access the files:

[MasterVol]<Susan>Todays>work
[MasterVol]<Susan>Yesterdays>work

as either:

<Susan>Todays>work
<Susan>Yesterdays>work

14-8

if just the volname is omitted, or:

        Todays>work
        Yesterdays>work

if the default volname and default dirname are
omitted.   <dirname> cannot be omitted unless
[volname] is also omitted.

If a SetPrefix operation is issued with the
default file prefix of Todays>, in addition to
the default volname and dirname established by
the SetPath operation above, the user can access
the files:

        [MasterVol]<Susan>Todays>work
        [MasterVol]<Susan>Yesterdays>work

as:

        work

and:

        <Susan>Yesterdays>work

The file in the last example above could no
longer be specified as:

        Yesterdays>work

because the file accessed would have been:

        [MasterVol]<Susan>Todays>Yesterdays>work

which was not what was meant.

Passwords

Password protection is available at three levels:

        volume,

        directory, or

        file.

A volume password protects a volume. A directory
password protects a directory on a volume.   A
file password protects a file in a directory on a
volume.

Volume passwords are specified with the IVolume utility. Directory passwords are specified in the CreateDir operation. File passwords are specified in the SetFileStatus operation.

Volume, directory and file passwords can consist of all alphanumeric characters, plus the period, ".", and the hyphen, "-". A volume, directory, or file password can have a maximum of 12 characters.

A file can be accessed by knowledge of its volume, directory, or file password. Knowledge of a volume password allows access to all the directories and all the files of that volume. Knowledge of a directory password allows access to all the files of that directory. Knowledge of a file password permits access that is dependent on the file protection level specified for that file. (See "File Protection" below.)

The OpenFile operation accepts a single password. This password is compared first against the volume password, then against the directory password, and last against the file password (if one was specified). Access is granted to open the file if any of these comparisons match.

The CreateFile operation accepts a single password that authorizes the creation of a file in the specified directory. It is not a password to be assigned to the file being created. This password is compared first against the volume password and then against the directory password. Access is granted to create the file if either of these comparisons match. (The SetFileStatus operation assigns a password to the file being created. The CreateDir operation assigns a password to the directory being created.)

A default password can be specified in the SetPath operation. It is used whenever an explicit password is not specified to an operation. The default password, like an explicit one, is used in a comparison against the volume, directory, and file passwords (in that order).

File Protection

A file is assigned a <u>file protection level</u>. A file protection level specifies the access

allowed to a file when the accessing process does not present a valid volume or directory password.

A default file protection level is specified for the files of a directory when it is created with the CreateDir operation. When a file is created, it is assigned the default file protection level of the directory in which it is created. The file protection level of a file can be changed with the SetFileStatus operation.

The file protection levels are described in Table 14-1 below. Three levels (unprotected, modify protected, and access protected) ignore file passwords; five levels (modify password, access password, read password, nondirectory modify password, and nondirectory access password) use file passwords.

The unprotected level is used for files that any process can read or modify.

The modify protected, modify password, and nondirectory modify password levels are used for files that any process can read but for which a password is needed to modify.

The access protected, access password, read password, and nondirectory access password levels are used for files that need a password to read or modify.

Table 14-1. File Protection Levels

Level	Decimal Value	Description
unprotected	15	The file is unprotected. It can be read or modified without a password.
modify protected	5	The file is modify protected. It can be read without a password. A volume or directory password is needed to modify it.
access protected	0	The file is read and modify protected. A

Table 14-1. File Protection Levels (Cont.)

Level	Decimal Value	Description
		volume or directory password is needed to read or modify it.
modify password	7	The file is modify protected. It can be read without a password. A password (volume, directory, or file) is needed to modify it.
access password	3	The file is read and modify protected. A password (volume, directory, or file) is needed to read or modify it.
read password	1	The file is read and modify protected. A password (volume, directory, or file) is needed to read it. A volume or directory password is needed to modify it.
nondirectory modify password	23	The file is modify protected. It can be read without a password. A volume or file password is needed to modify it; a directory password alone is insufficient.
nondirectory access password	19	The file is read and modify protected. A password (volume, directory, or file) is needed to read it. A volume or file password is needed to modify it; a directory password alone is insufficient.

CREATING AND ACCESSING A FILE

The file management system provides random access to 512-byte sectors of a file. (512 bytes is the size of a physical disk sector.) The operations of the file management system allow reading and writing of multiple sectors, starting with a particular sector of a file. The file management system provides device independence by masking the device characteristics of the disk on which the file is located.

Logical File Address

A logical file address (lfa) is used to locate a particular sector of a file. It specifies the byte position within a file; that is, it is the number (the offset) that would be assigned to a byte in a file if all the bytes were numbered consecutively starting with 0. An lfa is a 32-bit unsigned integer that must be on a sector boundary and is therefore a multiple of 512. For example, the lfa of the third sector of a file is 1024.

The two high-order bits of the lfa are reserved as special indicators. Bit 31 is set to override normal system checks and is used to attempt access to defective disks. Bit 30 is set to suppress retry of input/output to recover from errors. For example, a program logging high-speed digitized wave forms that could accept badly written data but not the time required for retry, would specify an lfa of 40000400h to specify the third sector of a file with error retry suppressed. The returned status code reports errors in the normal way even when the special indicators are set.

File Handle

A file handle (fh) is a 16-bit integer that uniquely identifies an open file. It is returned by the OpenFile operation and is used to refer to the file in subsequent operations such as Read, Write, and DeleteFile.

A file handle can be long-lived or short-lived. It is set long-lived by an OpenFileLL or SetFhLongevity operation. Only a short-lived (normal) file handle is closed by a CloseAllFiles

operation or automatically when an application system terminates. A long-lived, as well as a short-lived, file handle is closed by an explicit CloseFile operation or by the CloseAllFilesLL operation.

Memory Address

A memory address, as referred to in input/output operations, is always a 32-bit address that consists of a 16-bit segment base and a 16-bit offset.

Using a File

There are three steps in using a file:

1. creating it,

2. opening it, and

3. reading and writing it.

Creating a File

The following steps occur when a CreateFile operation is requested:

1. The OS verifies that a volume of the requested name is already online. (The Volume Home Block is brought into memory when a volume is placed online.)

2. The OS verifies that a directory of the requested name is on that volume. (The most recently used directory information is retained in memory.)

3. The OS verifies that a file of the requested name does not exist in that directory.

4. The OS allocates a File Header Block and assigns the requested number of disk sectors by using the Allocation Bit Map.

5. The OS inserts an entry for the file in the requested directory.

Opening a File

The following steps occur when an OpenFile operation is requested:

1. The OS verifies that a volume of the requested name is already online. (The Volume Home Block is brought into memory when a volume is placed online.)

2. The OS verifies that a directory of the requested name is on that volume. (The most recently used directory information is retained in memory.)

3. The OS verifies that a file of the requested name is in that directory.

4. The OS allocates a File Control Block, one or more File Area Blocks, and a pointer in the User Control Block to the File Control Block. (These structures are discussed in "System Data Structures" below.)

5. The OS copies the information from the File Header Block to the File Control Block and the one or more File Area Blocks.

6. The OS returns a file handle to the user process. The file handle serves to identify this particular File Control Block.

Reading and Writing a File

There are three ways to read and write the sectors of a file:

o   with the Read and Write procedures,

o   with the ReadAsync and CheckReadAsync and WriteAsync and CheckWriteAsync procedures, and

o   with a user-constructed request block and the Request and Wait (or Check) primitives.

The Read and Write procedures are the simplest way of doing input/output because much of the necessary housekeeping (for example, constructing a request block) and issuing the Request and Wait primitives is done automatically.   These two

procedures do not provide for any overlap between input/output operations and computation.

The ReadAsync and WriteAsync procedures are a more complex way of doing input/output. These two procedures allow a process to initiate an input/output transfer and then compute and/or initiate other input/output transfers before checking (with the CheckReadAsync and CheckWriteAsync procedures) for the successful completion of the transfer.

The user-constructed request block and the Request and Wait primitives are the most complex way of doing input/output. They allow the program to overlap multiple input/output operations and computation in an arbitrarily complex manner.

Local File System

A cluster workstation can have its own local file system. The local file system allows a cluster workstation to access files on local mass storage as well as files on mass storage at the master workstation. The file system filter process of the cluster workstation intercepts each file access request and directs it either to the local file system or to the master workstation.

When a request to open a file is intercepted, the filter process first routes it to the local file system. If the volume is not found, the request is routed to the master workstation.

The user can explicitly route a file access request to the master workstation by including the special character (!) before the volume specification.

Files on mass storage at the master workstation can be accessed by any cluster workstation. However, files on local mass storage cannot be accessed from the master workstation or from other cluster workstations. A local file must be copied to the master workstation if it is to be processed by the master workstation, another workstation in the cluster, or another node.

A local file must be copied to the master workstation before it can be processed by any of the following:

printer spooler,
batch manager,
RJE,
ISAM, or
any system service executing at the master
workstation or another cluster workstation

The file system filter process of the cluster
workstation duplicates the following master
workstation information:

default path information (specified in the
SetPath operation). This allows the GetUCB
operation to be serviced in the cluster
workstation.

date/time information (specified in the
SetDateTime operation).

A cluster workstation bootstrapped from its local
file system is a self-contained entity that must
access the master workstation only for shared
files. If a malfunction occurs at the master
workstation, the cluster workstation can continue
to operate normally provided all file accesses
are to local mass storage.

OPERATIONS: PROCEDURES AND SERVICES

File management operations are categorized by function in Table 14-2 below.

Table 14-2. File Management Operations by Function

Allocation	Defaults
ChangeFileLength	ClearPath
CreateFile	SetPath
DeleteFile	SetPrefix
Access	Directory
ChangeOpenMode	CreateDir
CloseAllFiles	DeleteDir
CloseAllFilesLL	GetDirStatus
CloseFile	ReadDirSector
OpenFile	SetDirStatus
OpenFileLL	
	Other
Input/Output	GetFhLongevity
	GetFileStatus
CheckReadAsync	GetUCB
CheckWriteAsync	QueryWSNum
Read	RenameFile
ReadAsync	SetFhLongevity
Write	SetFileStatus
WriteAsync	

Allocation

ChangeFileLength   expands or contracts an open file to a new length.

CreateFile   creates a file of the specified name in the specified directory on the specified volume.

DeleteFile   deletes an open file.

Access

ChangeOpenMode   allows a program to change the access mode of a file that is already open.

CloseAllFiles   closes all files that are currently open for the user, except those marked long-lived.

	CloseAllFilesLL	closes all files that are currently open for the user, including those marked long-lived.
	CloseFile	closes an open file.
	OpenFile	opens an already existing file, and returns a file handle.
	OpenFileLL	opens an already existing file, and returns a file handle marked long-lived.
Input/Output		
	CheckReadAsync	waits for input completion, checks the status code, and obtains the byte count of data read after a ReadAsync procedure.
	CheckWriteAsync	waits for output completion, checks the status code, and obtains the byte count of data written after a WriteAsync procedure.
	Read	transfers an integral number of 512-byte sectors from disk to memory.
	ReadAsync	initiates the transfer of an integral number of 512-byte sectors from disk to memory. The CheckReadAsync procedure must be called to check the completion status of the transfer.
	Write	transfers an integral number of 512-byte sectors from memory to disk.
	WriteAsync	initiates the transfer of an integral number of 512-byte sectors from memory to disk. The CheckWriteAsync procedure must be called to check the completion status of the transfer.

Defaults

ClearPath	clears the defaults established by the SetPath and SetPrefix operations.
SetPath	establishes a default volume, a default directory, and a default password.
SetPrefix	establishes a default file prefix that is prefixed to the file name part of a file specification if that file specification does not have an explicit volume name or directory name.

Directory

CreateDir	creates a directory of the specified name on the specified volume.
DeleteDir	deletes an empty directory.
GetDirStatus	allows a user to determine information about a directory.
ReadDirSector	reads a 512-byte sector of the specified directory.
SetDirStatus	allows a user to change a directory's password or default protection level.

Other

GetFhLongevity	copies the requested information on the longevity of the file handle to the specified area.	
GetFileStatus	copies the requested status information to the specified area.	
GetUCB	copies the User Control Block for the current user number to the specified area.	
QueryWSNum	returns the user number of the application system in the partition.	
RenameFile	changes the file name and/or the directory name of an existing file. A file can be renamed to another directory on the same volume.	
SetFhLongevity	sets how long a file handle is to survive.	
SetFileStatus	copies the specified status information from the specified memory area to the File Header Block.	

ChangeFileLength

Description

The ChangeFileLength service expands or contracts the file length to a new length. The end-of-file pointer in the File Header Block is set to reflect the new length.

Procedural Interface

ChangeFileLength (fh, lfaNewFileSize): ErcType

where

fh   is a file handle returned from an
                    OpenFile operation. The file must
                    be open in modify mode.

lfaNewFileSize
                    is the new file size in bytes. It
                    must be a multiple of 512.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	13
12	fh	2	
14	lfaNewFileSize	4	

ChangeOpenMode

Description

The ChangeOpenMode service allows a program to change the access mode of a file that is already open.

Error Code "Access Denied" (219) is returned if the password supplied does not grant access to the file in the new mode. Error code "File in Use" is returned if the new mode is "mode modify" and other users have the file open (in "mode read").

Procedural Interface

ChangeOpenMode   (fh, pbPassword, cbPassword, newMode) : ErcType

where

fh   is a file handle for a currently open file.

pbPassword
cbPassword   describe a password which gives access to the file in the new mode.

newMode   is the new file open mode.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	2	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	215
12	fh	2	
14	newMode	2	
16	reserved	2	
18	pbPassword	4	
22	cbPassword	2	

CheckReadAsync

Description

After calling the ReadAsync procedure to initiate a read, the requesting process continues execution. When the process wants to synchronize with the asynchronous read (that is, wait for its completion), the process does a CheckReadAsync. The CheckReadAsync procedure waits for input completion, checks the status code, and obtains the byte count of data read.

Status code 248 ("Wrong pRq argument") is returned if the pRq argument does not match the one of the preceding ReadAsync procedure.

Procedural Interface

CheckReadAsync (pRq, psDataRet): ErcType

where

pRq   is the same memory address as given in the pRq argument of the ReadAsync procedure.

psDataRet   is the memory address of the word to which the count of bytes successfully read is to be returned.

Request Block

The ReadAsync and CheckReadAsync procedures are procedural interfaces to the Read operation. See the Read operation below.

CheckWriteAsync

Description

After calling the WriteAsync procedure to initiate a write, the requesting process continues execution. When the process wants to synchronize with the asynchronous write (that is, wait for its completion), the process does a CheckWriteAsync. The CheckWriteAsync procedure waits for output completion, checks the status code, and obtains the byte count of data written.

Status code 248 ("Wrong pRq argument") status code is returned if the pRq argument does not match the one of the preceding WriteAsync procedure.

Procedural Interface

CheckWriteAsync (pRq, psDataRet): ErcType

where

pRq   is the same memory address as given in the pRq argument of the WriteAsync procedure.

psDataRet   is the memory address of the word to which the count of bytes success-fully written is to be returned.

Request Block

The WriteAsync and CheckWriteAsync procedures are procedural interfaces to the Write operation. See the Write operation below.

ClearPath

Description

The ClearPath service clears the defaults established by the SetPath and SetPrefix operations.

Procedural Interface

ClearPath: ErcType

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	2

CloseAllFiles

Description

> The CloseAllFiles service closes all files that are currently open for the user, except those marked long-lived.

Procedural Interface

> CloseAllFiles: ErcType

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	19

CloseAllFilesLL

Description

The CloseAllFilesLL service closes all files that are currently open for the user, including those marked long-lived.

Procedural Interface

CloseAllFilesLL: ErcType

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	62

CloseFile

Description

The CloseFile service closes an open file.

Procedural Interface

CloseFile (fh): ErcType

where

fh is the file handle returned from an OpenFile operation.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	10
12	fh	2	

CreateDir

Description

The CreateDir service creates a directory of the specified name on the specified volume. The volume name can be defaulted to that specified in a previous SetPath operation.

Status code 240 ("Directory already exists") status code is returned if a directory of the specified name already exists.

Procedural Interface

```
CreateDir (pbDirSpec, cbDirSpec, pbVolPassword,
           cbVolPassword, pbDirPassword,
           cbDirPassword, cSectors,
           defaultFileProtectionLevel): ErcType
```

where

pbDirSpec
cbDirSpec   describe a character string of the form {node} [volname] dirname.

pbVolPassword
cbVolPassword
                   describe the volume password that authorizes the creation of the directory on the specified volume.

pbDirPassword
cbDirPassword
                   describe the directory password to be assigned to this directory.

cSectors   is the size of the directory in 512-byte sectors.

                   The number of directory entries per sector depends on the length of the file names of the files created in the directory. An approximate value for the cSectors argument can be computed by dividing the expected maximum number of files ever to be created in the directory by 15.

defaultFileProtectionLevel
                   is the default file protection level to be assigned to files in this directory. (See Table 14-1 above.)

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	17
12	reserved	2	
14	cSectors	2	
16	defaultFile-ProtectionLevel	2	
18	pbDirSpec	4	
22	cbDirSpec	2	
24	pbVolPassword	4	
28	cbVolPassword	2	
30	pbDirPassword	4	
34	cbDirPassword	2	

CreateFile

Description

The CreateFile service creates a file of the specified name in the specified directory on the specified volume.

CreateFile creates a file; it does not open it. The OpenFile operation opens a file after it is created.

Status code 224 ("File already exists") status code is returned if a file of the specified name already exists.

Procedural Interface

CreateFile (pbFileSpec, cbFileSpec, pbPassword, cbPassword, lfaFileSize): ErcType

where

pbFileSpec
cbFileSpec   describe a character string of the form {node}[volname]<dirname>file-name. The distinction between uppercase and lowercase in file specifications is not significant in matching file names.

pbPassword
cbPassword   describes a volume or directory password that authorizes the creation of a file in the specified directory. It is <u>not</u> a password to be assigned to the file being created. A password can be assigned to the file being created with the SetFileStatus operation.

lfaFileSize   is the file size in bytes. It must be a multiple of 512.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	5
12	reserved	2	
14	lfaFileSize	4	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassword	4	
28	cbPassword	`2	

DeleteDir

Description

The DeleteDir service deletes an empty directory. All the files must be deleted from a directory before it can be deleted.

Status code 241 ("Directory not empty") status code is returned if the directory is not empty.

Procedural Interface

DeleteDir (pbDirSpec, cbDirSpec, pbPassword,
        cbPassword): ErcType

where

pbDirSpec
cbDirSpec   describe a character string of the
                form node volname dirname.

pbPassword
cbPassword   describe the volume or directory
                password. Either password autho-
                rizes the deletion of the directory.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	18
12	reserved	6	
18	pbDirSpec	4	
22	cbDirSpec	2	
24	pbPassword	4	
28	cbPassword	2	

DeleteFile

Description

The DeleteFile service deletes an open file.

Procedural Interface

DeleteFile (fh): ErcType

where

fh   is the file handle returned from an
                OpenFile operation.   The file must
                be open in modify mode.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	6
12	fh	2	

GetDirStatus

Description

The GetDirStatus allows a user to determine information about a directory. Either a volume password or the directory password are required.

Error code "Bad Mode" (218) is returned if the statusCode is invalid.

Procedural Interface

GetDirStatus   (pbDirname, cbDirName, pbPassword, cbPassword, pStatusRet, statusCode, sStatusMax) : ErcType

where

pbDirName cbDirName	describe the directory name.
pbPassword cbPassword	describe a password which gives access to the directory.
statusCode	specifies the status code. Status items and their codes are:

Code	Item	Size (bytes)
0	Directory Size	4
1	invalid	
2	File Protection Level	2
3	Password*	13

* The first byte of a password item is the number (0-12) of characters in the password.

pStatusRet sStatusMax	describe the memory area to which the status information is returned.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	2	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	216
12	status code	2	
14	reserved	4	
18	pbDirName	4	
22	cbDirName	2	
24	pbPassword	4	
28	cbPassword	2	
30	pStatusRet	4	
32	sStatusMax	2	

GetFhLongevity

Description

The GetFhLongevity service copies the requested information on the longevity of the file handle to the specified area.

Procedural Interface

GetFhLongevity (fh, pCodeRet): ErcType

where

fh   is the file handle returned from an OpenFile operation. The file can be open in either read or modify mode.

pCodeRet   is the memory address of the word to which the longevity code is returned. If the code is 0, the file handle is short-lived; if it is 1, the file handle is long-lived.

Request Block

sCodeRet is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	31
12	fh	2	
14	reserved	4	
18	pCodeRet	4	
22	sCodeRet	2	2

GetFileStatus

Description

The GetFileStatus service copies the requested status information to the specified memory area. If the specified area is not large enough to hold the requested information, the information is truncated.

Procedural Interface

GetFileStatus (fh, statusCode, pStatusRet, sStatusMax): ErcType

where

fh	is the file handle returned from an OpenFile operation. To get the password, the file must be open in modify mode. If the file is open in read mode, the file password field in the File Header Block is erased. The password fields in the Volume Home Block and Device Control Block are always erased.
statusCode	specifies the status code. Status items and their codes are:

Code	Item	Size (bytes)
0	File length	4
1	File type	1
2	File protection level	1
3	Password*	13
4	Date/time of creation	4
5	Date/time last modified	4
6	End-of-file pointer	4
7	File Header Block	512
8	Volume Home Block	256
9	Device Control Block	100
10	Application-specific field** in the File Header Block	64

*The first byte of a password item is the number (0-12) of characters in the password.

**For example, this field is used by B20 Word Processor files.

14-39

```
pStatusRet
sStatusMax      describe  the  memory  area  to  which
                the status information is returned.
```

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8
12	fh	2	
14	statusCode	2	
16	reserved	2	
18	pStatusRet	4	
22	sStatusMax	2	

GetUCB

Description

The GetUCB service copies the User Control Block for the current user number to the specified area. If the specified area is not large enough to hold the requested information, the information is truncated.

The User Control Block is described in "System Data Structures" below.

Procedural Interface

GetUCB (pUcbRet, sUcbMax): ErcType

where

pUcbRet
sUcbMax   describe the memory area to which the User Control Block is copied.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	27
12	reserved	6	
18	pUcbRet	4	
22	sUcbMax	2	

OpenFile

Description

The OpenFile service opens an already existing
file and returns a file handle. The file handle
returned by OpenFile is used to refer to the file
in subsequent operations such as Read, Write, and
DeleteFile.

Procedural Interface

OpenFile (pFhRet, pbFileSpec, cbFileSpec,
         pbPassword, cbPassword, mode): ErcType

where

pFhRet   is the memory address of the word to
                   which the file handle is returned.

pbFileSpec
cbFileSpec   describe a character string of the
                   form   {node}[volname]<dirname>file-
                   name.   The   distinction   between
                   uppercase   and   lowercase   in   file
                   specifications is not significant in
                   matching file names.

pbPassword
cbPassword   describe   either   the   volume,
                   directory,   or   file   password   that
                   authorizes access to the specified
                   file.

mode   is   read   (shared)   or   modify
                   (exclusive).   This is indicated by
                   16-bit values representing the ASCII
                   constants "mr" (mode read) or "mm"
                   (mode   modify).   In   these   ASCII
                   constants, the first character (m)
                   is   the   high-order   byte   and   the
                   second   character   (r   or   m,
                   respectively)   is   the   low-order
                   byte.

                   Access   in   read   mode   permits   the
                   returned file handle to be used as
                   an argument only to the CloseFile,
                   CheckReadAsync,   Read,   ReadAsync,
                   GetFhLongevity,   GetFileStatus,   and
                   SetFhLongevity operations.

Access in modify mode, however, permits the returned file handle to be used as an argument to all operations that expect a file handle.

If the file is currently open in read mode, access in read mode is permitted but attempted access in modify mode causes the return of status code 220 ("File in use").

If the file is currently open in modify mode, attempted access in either read or modify mode causes the return of status code 220 ("File in use").

Request Block

sFhMax is the size of a file handle and is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	4
12	reserved	2	
14	mode	2	
16	reserved	2	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	
30	pFhRet	4	
34	sFhMax	2	2

OpenFileLL

Description

The OpenFileLL service opens an already existing file and returns a file handle. The file handle is marked long-lived and can therefore be closed by the CloseFile and CloseAllFilesLL operations, but not by the CloseAllFiles operation. The file handle returned by OpenFileLL is used to refer to the file in subsequent operations such as Read, Write, and DeleteFile.

Procedural Interface

OpenFileLL (pFhRet, pbFileSpec, cbFileSpec,
                pbPassword, cbPassword,
                mode): ErcType

where

pFhRet   is the memory address of the word to which the file handle is returned.

pbFileSpec
cbFileSpec   describe a character string of the form {node} [volname] <dirname>file-name. The distinction between uppercase and lowercase in file specifications is not significant in matching file names.

pbPassword
cbPassword   describe the volume, directory, or file password that authorizes access to the specified file.

mode   is read (shared) or modify (exclusive). This is indicated by 16-bit values representing the ASCII constants "mr" (mode read) or "mm" (mode modify). In these ASCII constants, the first character (m) is the high-order byte and the second character (r or m, respectively) is the low-order byte.

                   Access in read mode permits the returned file handle to be used as an argument only to the CloseFile,

CheckReadAsync, Read, ReadAsync, GetFhLongevity, GetFileStatus, and SetFhLongevity operations. Access in modify mode, however, permits the returned file handle to be used as an argument to all operations that expect a file handle.

If the file is currently open in read mode, access in read mode is permitted but attempted access in modify mode causes the return of status code 220 ("File in use").

If the file is currently open in modify mode, attempted access in either read or modify mode causes the return of status code 220 ("File in use").

Request Block

sFhMax is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	97
12	reserved	2	
14	mode	2	
16	reserved	2	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	
30	pFhRet	4	
34	sFhMax	2	2

QueryWSNum

Description

The QueryWSNum service returns the number of the
cluster workstation.   QueryWSNum returns 0 if
executed on a standalone workstation.

Procedural Interface

QueryWSNum (pWSNumRet): ErcType

where

pWSNumRet   is the memory address of a word to
              which the number of the cluster
              workstation is returned.

Request Block

sWSNumRet is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	61
12	reserved	6	
18	pWSNumRet	4	
22	sWSNumRet	2	2

Read

Description

The Read service transfers an integral number of 512-byte sectors from disk to memory. Read returns only when the requested transfer is complete. The ReadAsync and CheckReadAsync procedures are used to overlap computation and input/output transfer.

To accommodate programming languages in which Read is a reserved word, ReadFile is permitted as a synonym for the Read service.

Procedural Interface

Read (fh, pBufferRet, sBufferMax, lfa,
      psDataRet): ErcType

where

fh   is a file handle returned from an
                OpenFile operation. The file can be
                open in either read or modify mode.

pBufferRet   is the memory address of the first
                byte of the buffer to which the data
                is to be read. The buffer must be
                word aligned.

sBufferMax   is the count of bytes to be read
                into memory. It must be a multiple
                of 512.

lfa   is the byte offset, from the
                beginning of the file, of the first
                byte to be read. It must be a
                multiple of 512.

psDataRet   is the memory address of the word to
                which the count of bytes success-
                fully read is to be returned.

Request Block

ssDataRet is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	35
12	fh	2	
14	lfa	4	
18	pBufferRet	4	
22	sBufferMax	2	
24	psDataRet	4	
28	ssDataRet	2	2

ReadAsync

The ReadAsync procedure initiates the transfer of an integral number of 512-byte sectors from disk to memory. The CheckReadAsync procedure must be called to check the completion status of the transfer.

The information returned by Read with its psDataRet argument and ErcType status is obtained by CheckReadAsync.

Procedural Interface

ReadAsync (fh, pBufferRet, sBufferMax, lfa, pRq, exchangeReply): ErcType

where

fh   is a file handle returned from an OpenFile operation. The file can be open in either read or modify mode.

pBufferRet   is the memory address of the first byte of the buffer to which the data is to be read. The buffer must be word aligned.

sBufferMax   is the count of bytes to be read to memory. It must be a multiple of 512.

lfa   is the byte offset, from the beginning of the file, of the first byte to be read. It must be a multiple of 512.

pRq   is the memory address of a 64-byte area to be used as workspace by ReadAsync.

exchangeReply
               is an exchange provided by the client process for the exclusive use of ReadAsync and CheckReadAsync.

Request Block

The ReadAsync and CheckReadAsync procedures are procedural interfaces to the Read operation. See the Read operation above.

ReadDirSector

Description

The ReadDirSector service reads a 512-byte sector
of the specified directory. ReadDirSector is
used primarily by the B20 Executive.

Procedural Interface

    ReadDirSector (pbDirSpec, cbDirSpec, pbPassword,
                   cbPassword, iSector,
                   pBufferRet): ErcType

    where

    pbDirSpec
    cbDirSpec   describe a character string of the
                   form {node}[volname]dirname.

    pbPassword
    cbPassword   describe the volume or directory
                   password. <u>Either</u> password autho-
                   rizes access to the directory
                   sector.

    iSector   is the number of the sector to be
                   read within the directory.

    pBufferRet   is the memory address of the first
                   byte of the buffer to which the data
                   is to be read. The buffer must be
                   word aligned.

Request Block

sBufferMax is always 512.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	25
12	reserved	2	
14	iSector	2	
16	reserved	2	
18	pbDirSpec	4	
22	cbDirSpec	2	
24	pbPassword	4	
28	cbPassword	2	
30	pBufferRet	4	
34	sBufferMax	2	512

RenameFile

Description

The RenameFile service changes the file name
and/or the directory name of an existing file. A
file can be renamed to another directory on the
same volume. However, it cannot be moved to
another node or volume by renaming it.

Procedural Interface

RenameFile (fh, pbNewFileSpec, cbNewFileSpec,
        pbPassword, cbPassword): ErcType

where

fh   is a file handle returned from an
                OpenFile operation. The file must
                be open in modify mode.

pbNewFileSpec
cbNewFileSpec
                describe a character string of the
                form   {node}[volname]<dirname>file-
                name.   The distinction between
                uppercase and lowercase in file
                specifications is not significant in
                matching file names.

pbPassword
cbPassword   describe a volume or directory
                password that authorizes the
                insertion of a file in the specified
                directory. It is <u>not</u> a password to
                be assigned to the file being
                renamed.   The SetFileStatus
                operation can be used to assign a
                password to the file being renamed.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	7
12	fh	2	
14	reserved	4	
18	pbNewFileSpec	4	
22	cbNewFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	

SetDirStatus

Description

The SetDirStatus allows a user to change a directory's password or default file protection level. A volume or directory password is required.

Error code "Bad Mode" (218) is returned if the statusCode is invalid.

Procedural Interface

SetDirStatus   (pbDirName, cbDirName, pbPassword, cbPassword, pStatus, statusCode, sStatus) : Erctype

where

pbDirName
cbDirName   describe the directory name.

pbPassword
cbPassword   describe a password which gives access to the directory.

statusCode   specifies the status code. Status items and their codes are:

Code	Item	Size (bytes)
0	invalid	
1	invalid	
2	File Protection Level	2
3	Password	*

* The length of the password is defined by sStatus, which must be less than or equal to 12.

pStatus
sStatus   describe the memory area from which the status information is copied.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	2	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	217
12	statusCode	2	
14	reserved	4	
18	pbDirName	4	
22	cbDirName	2	
24	pbPassword	4	
28	cbPassword	2	
30	pStatus	4	
32	sStatus	2	

SetFhLongevity

Description

The SetFhLongevity service sets how long a file handle is to survive. If the file handle is marked short-lived (the default condition when a file is first opened), it is closed by the CloseAllFiles, Exit, ErrorExit, and Chain operations. If it is marked long-lived, it is closed only by an explicit CloseFile operation or by a CloseAllFilesLL operation.

Procedural Interface

SetFhLongevity (fh, iCode): ErcType

where

fh   is the file handle returned from an OpenFile operation. The file can be open in either read or modify mode.

iCode   is either 0 for a short-lived file handle or 1 for long-lived one.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	30
12	fh	2	
14	iCode	2	

SetFileStatus

Description

The SetFileStatus service copies the specified status information from the specified memory area to the File Header Block for the file defined by the file handle. SetFileStatus cannot change the file length. The ChangeFileLength operation can be used to change the file length.

Procedural Interface

SetFileStatus (fh, statusCode, pStatus,
                sStatus): ErcType

where

fh   is a file handle returned from an OpenFile operation. The file must be open in modify mode.

statusCode   specifies the status code. Status items and their codes are:

Code	Item	Size (bytes)
1	File type	1
2	File protection level	1
3	Password	*
4	Date/time of creation	4
5	Date/time last modified	4
6	End-of-file pointer	4
7	invalid	
8	invalid	
9	invalid	
10	Application-specific field** in the File Header Block	64

*The length of password is defined by sStatus, which must be less than or equal to 12.

**This field is used by B20 Word Processor files, for example.

pStatus
sStatus   describe the memory area from which the status information is copied.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	9
12	fh	2	
14	statusCode	2	
16	reserved	2	
18	pStatus	4	
22	sStatus	2	

SetPath

Description

The SetPath service establishes a default volume, a default directory, and a default password. It also clears the default file prefix. A subsequent ClearPath operation clears the defaults.

Procedural Interface

```
SetPath (pbVolSpec, cbVolSpec, pbDirName,
        cbDirName, pbPassword,
        cbPassword): ErcType
```

where

pbVolSpec
cbVolSpec   describe the default volume specifi-
cation of the form {node}[volname].

pbDirName
cbDirName   describe the default directory name.

pbPassword
cbPassword   describe the default password.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	1
12	reserved	6	
18	pbVolSpec	4	
22	cbVolSpec	2	
24	pbDirName	4	
28	cbDirName	2	
30	pbPassword	4	
34	cbPassword	2	

SetPrefix

Description

The SetPrefix service establishes a default file prefix that is prefixed to the file name part of a file specification if that file specification does not have an explicit volume name or directory name. A new SetPrefix overrides a previous SetPrefix. The default prefix established by SetPrefix can be removed by:

1. another SetPrefix that specifies a null string,

2. the SetPath operation, or

3. the ClearPath operation.

Procedural Interface

SetPrefix (pbPrefix, cbPrefix): ErcType

where

pbPrefix
cbPrefix   describe the character string that is to be used as a default file prefix.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	3
12	reserved	6	
18	pbPrefix	4	
22	cbPrefix	2	

Write

Description

The Write operation transfers an integral number of 512-byte sectors from memory to disk. Write returns only when the requested transfer is complete. The WriteAsync and CheckWriteAsync procedures are used to overlap computation and input/output transfer. Write can also be accessed as the WriteFile operation.

Attempting to write beyond the end of a file results in the return of status code 2 ("End of medium").

To accommodate programming languages in which Write is a reserved word, WriteFile is permitted as a synonym for the Write service.

Procedural Interface

```
Write (fh, pBuffer, sBuffer, lfa,
       psDataRet): ErcType
```

where

fh   is a file handle returned from an OpenFile operation. The file must be open in modify mode.

pBuffer   is the memory address of the first byte of the buffer from which the data is to be written. The buffer must be word aligned.

sBuffer   is the count of bytes to be written from memory. It must be a multiple of 512.

lfa   is the byte offset, from the beginning of the file, of the first byte to be written. It must be a multiple of 512.

psDataRet   is the memory address of the word to which the count of bytes successfully written is to be returned.

Request Block

ssDataRet is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	36
12	fh	2	
14	lfa	4	
18	pBuffer	4	
22	sBuffer	2	
24	psDataRet	4	
28	ssDataRet	2	2

WriteAsync

Description

The WriteAsync procedure initiates the transfer of an integral number of 512-byte sectors from memory to disk. The CheckWriteAsync procedure must be called to check the completion status of the transfer.

The information returned by Write with its psDataRet argument and ErcType status is obtained by CheckWriteAsync.

Procedural Interface

WriteAsync (fh, pBuffer, sBuffer, lfa, pRq,
            exchangeReply): ErcType

where

fh   is a file handle returned from an
                OpenFile operation. The file must
                be open in modify mode.

pBuffer   is the memory address of the first
                byte of the buffer from which the
                data is to be written. The buffer
                must be word aligned.

sBuffer   is the count of bytes to be written
                from memory. It must be a multiple
                of 512.

lfa   is the byte offset, from the
                beginning of the file, of the first
                byte to be written. It must be a
                multiple of 512.

pRq   is the memory address of a 64-byte
                area to be used as workspace by
                WriteAsync.

exchangeReply
                is an exchange provided by the
                client process for the exclusive use
                of WriteAsync and CheckWriteAsync.

Request Block

The WriteAsync and CheckWriteAsync procedures are procedural interfaces to the Write operation. See the Write operation above.

VOLUME CONTROL STRUCTURES

A disk volume contains volume control structures after it is initialized with the IVolume utility. (See the B20 System Software Operation Guide, form 1148772.) These structures allow the file management system to manage (allocate, deallocate, locate, avoid duplication of) the space on the volume not already allocated to the volume control structures themselves.

The volume control structures include:

o   the Volume Home Block,

o   the File Header Blocks,

o   the Master File Directory,

o   the directories, and

o   the Allocation Bit Map, among others.

There are duplicate Volume Home Blocks (working and initial copies) and (normally) duplicate File Header Blocks (primary and secondary copies) on the volume for reliability. The primary and secondary copies of the File Header Blocks are located on different cylinders and at different rotational positions and are accessed (except for floppy disks) by different read/write heads. These duplicates ensure that damage to one copy does not cause a loss of data. The IVolume utility permits suppression of duplicate File Header Blocks. However, this reduces reliability and is not recommended.

The initial copy, unlike the working copy, of the Volume Home Block, is not modified after it is created. However, the primary and secondary copies of the File Header Blocks are always true duplicates.

The location on the volume of the volume control structures minimizes disk arm movement. In particular, the structures that are necessary to create and open files (the working copy of the Volume Home Block, the File Header Blocks, the Master File Directory, the directories, and the Allocation Bit Map) are located near one another and near the middle of the disk. The initial copy of the Volume Home Block is located near the start of the disk. Both the primary and

secondary copies of the File Header Blocks are
located in a single area, the disk address and
size of which are recorded in the working and
initial copies of the Volume Home Block.

Figure 14-1 shows the interrelationships of the
volume control structures.

Volume Home Block

There is a <u>Volume Home Block</u> (VHB) for each
volume. The VHB is the root structure (that is,
the starting point for the tree structure) of
information of a disk volume. The VHB contains
information about the volume such as its name and
the date it was created. The VHB also contains
pointers to the Allocation Bit Map, the Bad
Sector File, the File Header Blocks, the Master
File Directory, the directories, the System
Image, the Crash Dump Area, and the Log File.
The VHB is 1 sector in size. (See Table 14-3 on
the Volume Home Block below.)

Allocation Bit Map and Bad Sector File

The <u>Allocation Bit Map</u> controls the assignment of
disk sectors. It has 1 bit for every sector on
the disk and the bit is set if the sector is
available. The size of the Allocation Bit Map
depends on the size of the volume. If a sector
of a disk is unusable, there is an entry in the
<u>Bad Sector File</u>. The Bad Sector File is 1 sector
in size.

File Header Block

There is a <u>File Header Block</u> (FHB) for each
file. The FHB of each file contains information
about that file such as its name, password,
protection level, the date/time it was created,
the date/time it was last modified, and the disk
address and size of each of its Disk
Extents. The FHB is 1 sector in size. (See
Table 14-4 on the File Header Block below.)

Disk Extent

A <u>Disk Extent</u> is one or more contiguous disk
sectors that compose all or part of a file. The
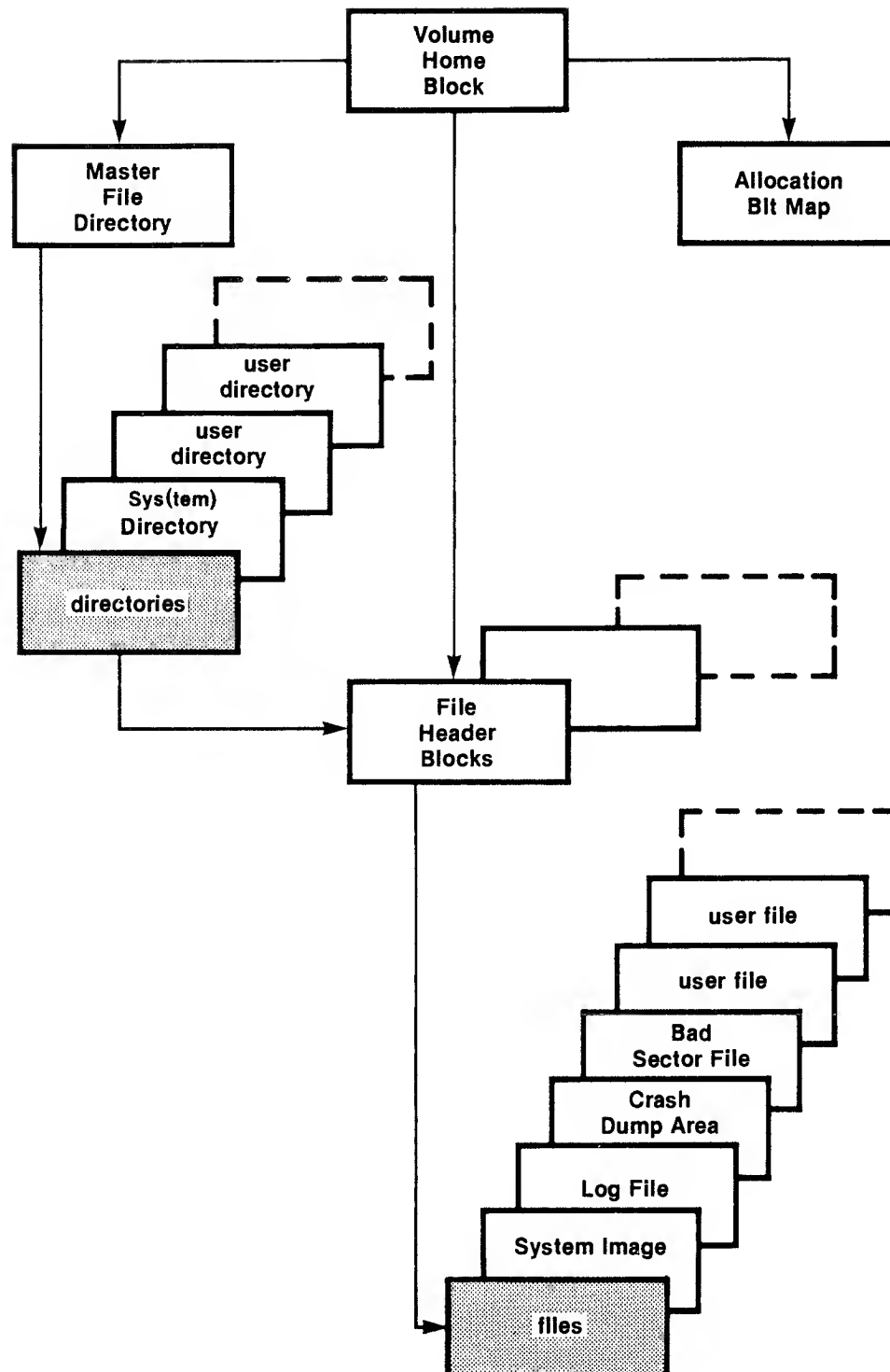
Figure 14-1. Volume Control Structures

entry for the Disk Extent in the FHB is 8
bytes: 4 bytes specify its location and 4 bytes
specify its size.

Table 14-3. Volume Home Block

Offset	Field	Size (bytes)	Description
0	checksum	2	
2	lfaSysImageBase	4	
6	cPagesSysImage	2	
8	lfaBadBlkBase	4	
12	cPagesBadBlk	2	
14	lfaCrashDumpBase	4	
18	cPagesCrashDump	2	
20	volName*	13	
33	volPassword*	13	
46	lfaVhb	4	
50	lfaInitialVhb	4	
54	creationDT	4	
58	modificationDT	4	
62	lfaMfdBase	4	
66	cPagesMfd	2	
68	lfaLogBase	4	
72	cPageLog	2	
74	currentLogPage	2	
76	currentLogByte	2	
78	lfaFileHeadersBase	4	
82	cPagesFileHeader	2	
84	altFileHeaderPageOffset	2	
86	iFreeFileHeader	2	
88	cFreeFileHeaders	2	
90	clusterFactor	2	
92	defaultExtend	2	
94	allocSkipCnt	2	
96	lfaAllocBase	4	
100	allocPageCnt	2	
102	lastAllocPg	2	
104	lastAllocWd	2	
106	lastAllocBit	2	
108	cFreePages	4	
112	idev	2	
114	rgLruDirEntries (sRgLruDirEntries)	105	
219	magicWd	2	

*The first byte contains the character count.

14-68

Offset	Field	Size (bytes)	Description
221	BootBaseSector	1	
222	BootBaseHead	1	
223	BootBaseCyl	2	
225	BootMaxPageCount	2	These fields describe for the Bootstrap ROM the location and file size of the program to be bootstrapped.
227	BadBlkBaseSector	1	
228	BadBlkBaseHead	1	
229	BadBlkBaseCyl	2	
231	BadBlkMaxPgCnt	2	These fields describe the bad block map used by IVolume when reinitalizing a volume.
233	DumpBaseSector	1	
234	DumpBaseHead	1	
235	DumpBaseCyl	2	
237	DumpMaxPageCount	2	These fields describe the location and file size of a crash dump area to be used by the Bootstrap ROM.
239	bytesPerSector	2	
241	sectorsPerTrack	2	
243	tracksPerCyl	2	

Offset	Field	Size (bytes)	Description
245	cylindersPerDisk	2	These fields describe the disk size parameters.
247	interleaveFactor	1	
248	sectorSize	2	
250	spiralFactor	1	
251	startingSector	1	These fields describe formatting parameters by IVolume.
252	reserved	4	Reserved for future expansion.

Table 14-4. File Header Block

Offset	Field	Size (bytes)
0	checksum	2
2	fileHeaderPageNum	2
4	fileName*	51
55	password*	13
68	dirName*	13
81	fileHeaderNum	2
83	extensionHeaderNumChain	2
85	headerSequenceNum	1
86	fileClass	1
87	accessProtection	1
88	lfaDirPage	4
92	creationDT	4
96	modificationDT	4
100	accessDT	4
104	expirationDT	4
108	fNoSave	1
109	fNoDirPrint	1
110	fNoDelete	1
111	lfaEndOfFile	4
115	defaultExpansion	4
119	freeRunIndex	2
121	vda(runsPerFhb)	128
249	runLength(runsPerFhb)	128
377	(reserved for expansion)	71
448	application-specific field	64

*The first byte contains the character count.

BootExt.Sys

BootExt.Sys is placed in the Sys directory of every B22 volume which has a Sysimage.Sys file specified. The file is 10 sectors long, and is pointed to in the VHB Boot ROM fields starting at offset 221.

The first stage bootstrap program is automatically placed in BootExt.Sys by IVolume during initialization by copying from [Sys]<Sys>BootExt.Sys. [Sys] is [f0] if you have booted from a floppy, or [d0] if you have booted from the hard disk.

Extension File Header Block

An FHB can accommodate 32 Disk Extents. A file
that contains more than 32 Disk Extents requires
extension File Header Blocks. Extension FHBs are
seldom necessary unless the user places an
unusually heavy burden on the file management
system by, for example, expanding the same file
many times or fragmenting the available disk
space by frequently deleting and creating files
on a nearly full volume that is seldom refreshed.
(A volume is refreshed by using the Backup
Volume, IVolume, and Restore utilities. See the
B20 System Software Operation Guide, form 1148772
for more details about these utilities.)

Master File Directory and Directories

There is an entry for each directory on the volume in the <u>Master File Directory</u> (MFD), including the Sys Directory (see below). The position of an entry within the MFD is determined by randomization (hashing) techniques. The entry contains the directory's name, password, location, and size. (See Table 14-5 on the Entry for a Directory in the MFD below.)

Table 14-5. Entry for a Directory in the Master File Directory

Offset	Field	Size (bytes)
0	dirEntryName*	13
13	password*	13
26	lfaBase	4
30	cPages	2
32	defaultAccessCode	1
33	lruCnt	2

*The first byte contains the character count.

There is an entry for each file in one of the <u>directories</u> on the volume. The position of an entry within a directory is determined by randomization (hashing) techniques. The entry contains the file's name and a pointer to the File Header Block.

The MFD and the directories provide fast access to the File Header Block of a specific file. They do not, however, contain any information about the file that is not also contained in its File Header Block. (The most recently used directory information is retained in memory.)

System Directory

The <u>Sys</u>(tem) <u>Directory</u> is different from other directories in two ways. First, when a volume is initialized, its MFD contains only one entry and that is for the Sys Directory. (The other directories are created by the CreateDir

operation.)   Second, the Sys Directory contains entries for all system files.   These files must not be deleted, renamed, or overwritten.

These file entries are required in the Sys Directory of each volume:

o   the Bad Sector File (BadBlk.Sys),

o   the Master File Directory (Mfd.Sys), and

o   the File Header Blocks (FileHeaders.Sys).

SYSTEM VOLUME

The Sys(tem) Directory of the Sys(tem) Volume contains entries for system files that are not necessary in the Sys Directories of other volumes. These additional entries must be placed in [Sys]<Sys> when the volume is initialized. SysImage.Sys, CrashDump.Sys, and Log.Sys are created (but not initialized) by the IVolume utility. The other file entries are created using the CreateDir operation or the Create Directory command (see the B20 System Executive Reference Manual, form 1144474).

These system files are:

the System Images (SysImage.Sys and WS-nnn>SysImage.Sys),

the Crash Dump Areas (CrashDump.Sys and WSnnn>CrashDump.Sys),

the Log File (Log.Sys),

the standard character font (Sys.Font).

For information on other initialization files, including the Executive and Debugger, see the Release Notice for the current OS version and the section on "Getting Started" in the B20 System Programmers and Assembler Reference Manual (Part 1), form 1148699.

System Image

The System Image (the file SysImage.Sys) contains a run-file copy of the OS for the standalone or master workstation.

In a cluster system, the OS for the cluster workstations also must be placed in this volume and directory in the file WSnnn>SysImage.Sys where nnn is the workstation type, as follows:

000	B22
255	B21-1
254	B21-3
253	B21-4

If the file WSnnn>SysImage.Sys does not exist, WS>SysImage.Sys is reset.

See the B20 System Programmers and Assembler Reference Manual (Part 1), form 1148699 for more detail.

Crash Dump Area

The <u>Crash Dump Area</u> (the file CrashDump.Sys) contains a binary memory dump for the standalone or master workstation in the event of a system failure.

The files WSnnn>CrashDump.Sys (if they exist) contain binary memory dumps for cluster workstations in the event of system failures at the cluster workstations. In the file specification, nnn is the workstation identification of the cluster workstation. (Crash Dump files are created with the CreateFile operation.)

If the files WSnnn>CrashDump.Sys do not exist, the memory dump is made to the file WS>CrashDump.Sys (if it exists). This file eliminates the need for a Crash Dump file for each cluster workstation.

Log File

The Log File (the file Log.Sys) is an error-logging file. An entry is placed in the Log File for each recoverable and nonrecoverable device error. This file can be used as a general-purpose logging file, for example, to write entries for accounting information for system services. The PLog utility (see the B20 System Software Operation Guide, form 1148772) prints the content of this file.

Standard Character Font

The standard character font is loaded from the file Sys.Font into the font RAM (except on a B21, which has the standard character font in ROM).

$ Directories

The $ Directories are special directories required for the software to operate correctly. When a request with the directory name of <$> is given as part of a file specification to the OS, the directory name is expanded to the form <$nnn>, where nnn is the user number of the application partition. This expansion occurs only if the directory name is <$>.

For example, the following file specifications are expanded as shown when they are part of a request from an application system in the primary application partition of a standalone or master workstation (user number 0):

[Vol]<$>Filename   to   [Vol]<$000>Filename

<$>Filename   to   <$000>Filename

[Vol]<$xyz>Filename   to   [Vol]<$xyz>Filename

If an application system in a cluster workstation (with user number 3, for example,) generates a request with a directory name of <$>, it is expanded as follows:

[Vol]<$>Filename   to   [Vol]<$003>Filename

All software that uses temporary files attempts to place those files in the [Sys]<$> directory first, and, if that fails, then in the logged-in volume and directory.

Since the user number(s) of a cluster workstation are reassigned whenever the system is bootstrapped, the $ directories should not be used for permanent files.

SYSTEM DATA STRUCTURES

System data structures are data areas contained within the OS and necessary for its operation. They are often configuration-dependent. The five system data structures related to the file management system are:

the User Control Block,

the File Control Block,

the File Area Block,

the Device Control Block, and

the I/O Block.

The User Control Block and the Device Control Block are user-accessible and are described below.

User Control Block

There is a User Control Block (UCB) for each user number. The UCB contains the default volume, default directory, default password, and default file prefix set by the last SetPath and SetPrefix operations. (See Table 14-6 on the User Control Block below.)

There is a user number for each application partition.

Table 14-6. User Control Block

Offset	Field	Size (bytes)
0	logInId	2
2	defaultVol*	13
15	defaultDir*	13
28	defaultPassword*	13
41	prefix*	41
54	verifyCode	1

*The first byte contains the character count.

UCBs reside in master and cluster workstations as discussed below.

User Control Blocks in the Master Workstation

Two types of UCBs reside in the master workstation:

local UCBs for secondary application partitions, and

remote UCBs for file access by cluster workstations.

Local UCBs are allocated for secondary application partitions created within the master workstation. They are associated with Batch Control Blocks and Partition Descriptors, and are statically allocated by the OS.

Remote UCBs are allocated for tasks located in cluster workstations that access files at the master workstation. They are dynamically allocated and deallocated by the OS.

User Control Blocks in the Cluster Workstations

Local UCBs are allocated in the cluster workstation for the local file system. They are associated with Batch Control Blocks and Partition Descriptors, and are statically allocated by the OS.

Device Control Block

There is a Device Control Block (DCB) for each physical device. The DCB contains information, generated at system build, about the device. For a disk, the information includes how many tracks are on a disk, the number of sectors per track, etc. The DCB points to a chain of I/O Blocks. (See Table 14-7 below.)

Table 14-7. Device Control Block

Offset	Field	Size (Bytes)	Description
(device independent fields)			
0	fMountable	1	True (OFFh) if the device may have a valid B20 file system.
1	fNonSharable	1	False (0) if the device may be shared by multiple users. True if the device is non-shareable, such as a tape device.
2	fDoubleDensity	1	True if the media is a diskette and is dual density.
3	fNoMultiTrack	1	True if the controller does not support multiple track read/write operations. This must be True for floppy drives on B21 and B22 with old controller.
4	fAttention	1	Device has come ready or not ready. Flag is set in interrupt routine and cleared by MassIO process upon Mount or Dismount.
5	fTimeout	1	Flag is set if a timeout has occurred.
6	devName(13)	13	The name of the device. The first byte is the size.
19	devpassword(13)	13	The device password. The first byte is the size.
32	reserved	1	

Offset	Field	Size (bytes)	Description
33	unitNum	1	The logical unit number of the device.
34	state	1	The state of the device where: 0 means idle, 1 means seeking, 2 means busy (reading or writing), and 3 means retrying.
35	unitStatus	1	The status of the unit, where: 0 means ready 1 means not ready or inoperable.
36	deviceClass	1	The device class where: 0 means Winchester on either a B21 or on a B22. 1 means a floppy on B21 or on B22.
37	userCount	1	The number of users that have opened the device or the number of files in file system on device.
38	oVhb	2	An offset pointer (from the OS's DGroup) to the Volume Home Block of the device.
40	oIobFirst	2	Offset to the head of the I/O Block list for the device.
42	oIobLast	2	Offset to the tail of the IOB list.
44	lfaMax	4	The size (in bytes) of the device.

Offset	Field	Size (bytes)	Description
48	lfaMask	2	Mask used for separating the upper bits of lfa which contain control information.
50	verifyKey	2	Used in creation of file handles for checking the type of resource which is open.
52	reserved	6	
58	softErrorCnt	2	
60	hardErrorCnt	2	Accumulated count of errors during an I/O.
62	currentCylinder	2	Set after every successful seek. Used by by queueing algorithm.
64	sectorSizeCode	1	Sector size, where:
			0 means use bytes per sector field (must be less than 256),
			1 means 256 bytes,
			2 means 512 bytes, and
			3 means 1024 bytes.
65	gapLength(2)	2	
67	dataLength	1	Floppy controller parameters.
68	bytes/Sector	2	
70	sectors/Track	2	
72	tracks/Cylinder	2	
74	cylinders/Disk	2	Disk size parameters.

SECTION 15
QUEUE MANAGEMENT

OVERVIEW

The queue management facility controls named, priority-ordered, disk-based queues. The files that contain these queues are called queue entry files. Each queue entry file contains information for a single type of processing, such as spooled printing, batch processing, or remote job entry (RJE). This information is created, accessed, and modified by both client processes and server processes such as the printer spooler, batch manager, or RJE. Because the queue entry files are disk-based, their contents are immune to system failures.

In a cluster configuration, the queue management facility must be installed at the master workstation. However, the server processes that use the queue management facility can be installed at cluster workstations as well as at the master workstation. Multiple server processes in different cluster workstations can serve the same queue simultaneously.

The system administrator defines the queues to be used in the system. Each queue is assigned as a unique name and a queue entry file specification.

Client processes can then add queue entries by using operations in which a queue entry file is referenced by a queue name. The client process need not specify the location of the server process. The first available server process in the cluster can serve the queue entry.

Figure 15-1 shows an example of a cluster configuration with the queue management facility, a client process, and a server process (printer spooler).

Master Workstation

Queue Manager

Queue Entry Files
Data Files

Cluster Workstation

Cluster Workstation

Cluster Workstation

Cluster Workstation

Requestor

Printer Spooler

Printer(s)
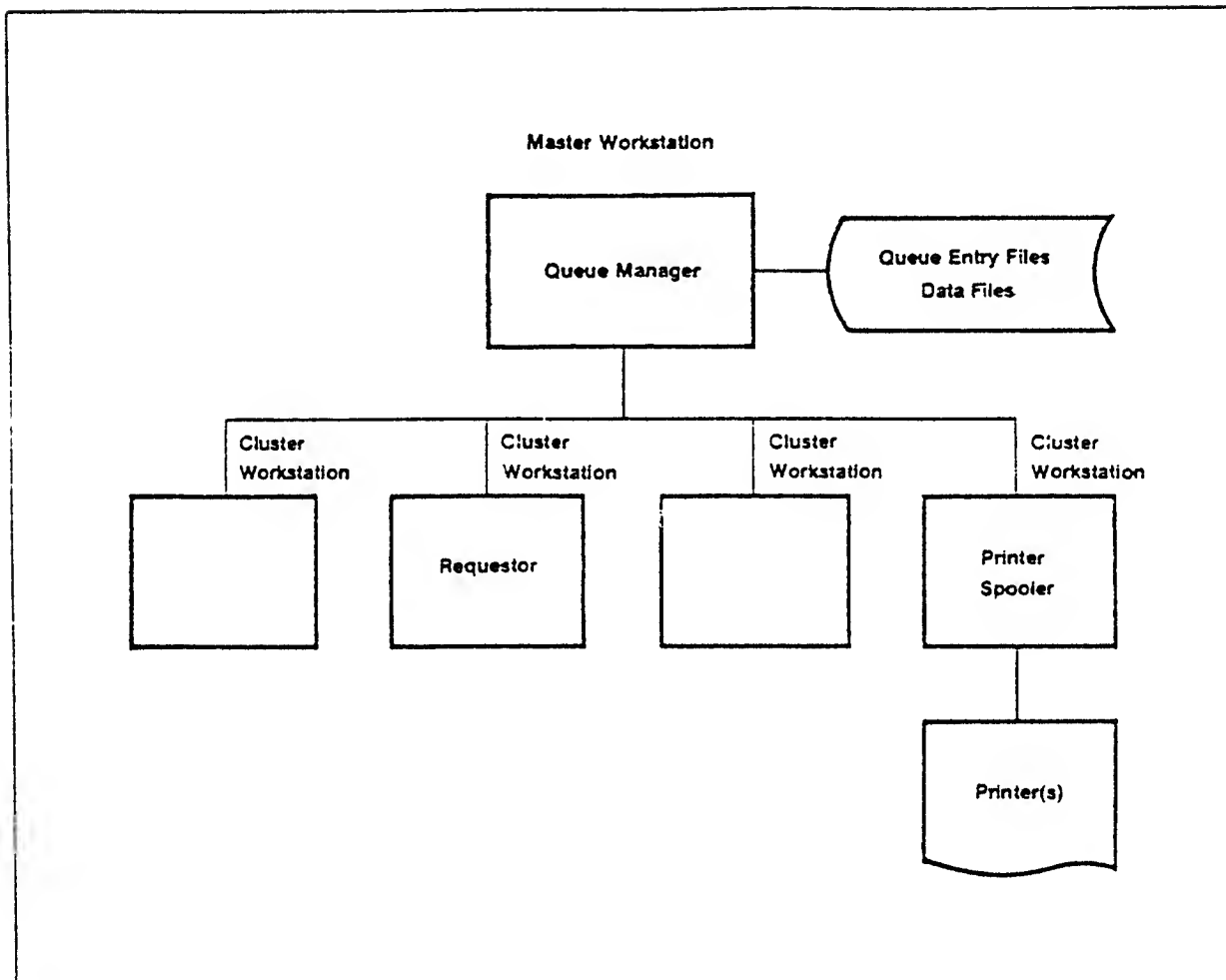
Figure 15-1. Example Configuration With Queue Management Facility

CONCEPTS

The queue management facility acts as a central switch between client and server processes.

Client Processes

Client processes submit requests for processing services, such as printing, transmission, and batch processing of files, to the queue manager. By using the queue management facility, client processes can:

access queue entry files by using operations that specify the queue name,

submit entries to the appropriate queue entry file,

delete previously queued entries, and

obtain a list of entries queued.

Server Processes

Server processes (such as the printer spooler, RJE, and batch manager) serve the queue entry files. The queue management facility allows server processes to:

specify the queue(s) (and therefore the queue entry files) they will serve,

process entries in the specified queue(s), and

request the removal of queue entries that are processed.

Sequence for Using Queue Management Facility

A simplified sequence for installing and using the queue management facility is described below.

1. The system administrator creates a queue index file in the master workstation. The queue index file is a system-wide text file that defines the queues to be used in the system. The queue index file assigns to each queue a queue entry file for storing queue entries submitted by client processes, the size of the queue entry, and the queue type.

2. The queue manager is installed in the master workstation with the Install Queue Manager utility. Typically, this is part of a submit or batch job control file when the system is bootstrapped.

3. After the queue manager is installed, it opens the queue entry files named in the queue index file. The queue entry files are maintained in the master workstation.

4. A server process (such as a printer spooler, RJE, or batch manager) wishing to serve a particular queue uses the EstablishQueueServer operation to establish itself as an active server.

5. A client process adds entries to the specified queue entry file with the AddQueueEntry operation.

6. The server process obtains for processing a particular queue entry with the MarkKeyedQueueEntry, or the next available queue entry with the MarkNextQueueEntry operation. The queue manager marks the queue entry as being in use to prevent other server processes from operating on it. The marked queue entry remains in the queue entry file until it is removed (see next step).

7. The server process services the marked queue entry, then removes the processed entry from the queue entry file with the RemoveMarkedQueueEntry operation.

8. When the server process no longer wishes to serve a queue, it removes itself from the list of active servers with the TerminateQueueServer operation.

Queue Index File

The queue index file is a system-wide text file that defines the queues to be used in the system. It contains information such as the name of each queue to be used in the system and the associated queue entry file.

The system administrator must create the queue index file [Sys]<Sys>Queue.Index in the master workstation.

The queue index file is created with the Text Editor or Word Processor. A record of the following format is required for each queue:

queueName/fileSpec/entrySize/queueType   RETURN

.

.

.

where

queueName   is a user-defined queue name that is unique to the installation. The name can be any name of up to 50 characters, except the following system device names: Comm, Kbd, Lpt, Nul, Ptr, Tape, Vid, and X25. Examples of acceptable names are: SpoolerA, SPL, PrinterX, BatchCarol, Centronix, Diablo, and RJEtoBoston.

fileSpec   is the file specification of the queue entry file in which queue entries submitted by client processes are stored (for example, [Winl]<Sys>SpoolerAQueueEntryFile).

entrySize   is the size of an entry for the queue entry file. The size is the number of 512-byte sectors per entry. For example, to define 1024-byte entries, specify an entry size of 2. In this case, 984 bytes are usable and 40 are reserved for the queue manager.

queueType   is the type of the queue (an integer less than or equal to 255), which enables a consistency check. The queue manager checks the type against the type in operations to add entries to the queue and in operations to establish servers for the queue. Types 0-80 are reserved; types 1, 2, and 3 are assigned as follows:

Type	Assignment
1	Printer Spooler queue
2	RJE queue
3	Batch queue

A sample queue index file is shown in Figure 15-2 below.

```
SpoolerA/SpoolerAQueueEntryFile/1/1 <RETURN>

RJEBoston/RJEBostonQueueEntryFile/1/2 <RETURN>

BatchCarol/BatchCarolQueueEntryFile/1/3 <RETURN>
```

Figure 15-2. Sample Queue Index File

The above example defines one queue entry file for each queue name. Each queue-oriented service generally requires more than one type of queue entry file. See Table 15-1 below.

Table 15-1. Examples of Queue Entry Files

Server Process	Type	Number Required
Batch manager	Scheduling	1 per batch queue
	Control	1 per batch manager
	Status	1 per cluster configuration
Printer spooler	Scheduling	1 per print class
	Control	1 per printer
	Status	1 per cluster configuration
RJE	Transmit	1 per cluster configuration
	Receive	1 per cluster configuration

Client processes add entries to the scheduling queue entry files. In the case of RJE, entries are added to the transmit queue entry file and removed from the receive queue entry file. The control and status queue entry files are used internally by the server processes for control and status purposes. For further information on queue entry files required in the queue index file, see:

the B 20 System Software Operation Guide, form 1148772 for the printer spooler, and

the B 20 2780; 3780 RJE Reference Manual, form 1148731 for the RJE.

Installing the Queue Manager

The queue manager is installed with the Install Queue Manager utility (see the B 20 System Software Operation Guide, form 1148772). Typically, this is part of a submit file or batch job control file that is executed when the system is bootstrapped. See the B 20 System Executive Reference Manual, form 1144474.

The queue manager can be installed either:

in an extended system partition, in which case the queue manager can be removed only when the OS is reloaded (see the "System Service Management" section), or

in a secondary application partition, in which case the queue manager can be removed with the TerminatePartitionTasks or VacatePartition operation (see the "Application Partition Management" section).

Queue Entry File

A queue entry file contains information for a single type of processing such as spooled printing, batch processing, or RJE.

More than one type of queue entry file is generally required for each queue-oriented service (for example, scheduling, control, and status queue entry files are required for a printer spooler queue). (See Table 15-1 above.)

The client process specifies the queue name when submitting a queue entry for processing. The queue entry is automatically placed in the appropriate queue entry file by the queue manager.

When the queue manager is installed, it opens the queue entry files specified in the queue index file. If a queue entry file does not exist, it is created with an initial size of 30 entries.

If a queue entry file has insufficient space for adding an entry, the queue manager expands that queue entry file by an increment sufficient to contain 30 entries.

Queue Entry

A queue entry is a formatted request for processing that is added to the specified queue entry file by client processes. Client and server processes communicate by way of fields within the queue entries located at fixed offsets known to both client and server processes. When a server process is available, it obtains a queue entry for processing.

A queue entry is a number of contiguous 512-byte sectors in a queue file. Each queue entry consists of two parts:

The first 40 bytes of the queue entry are reserved for the queue manager and include control information (see "Queue Status Block" below).

The remaining bytes are type-specific, that is, they are specific to the type of the queue (see "Sample Queue Entry" below).

CLIENT OPERATIONS

A client process can add entries to queue entry files, read the entries of a queue entry file (typically, to determine the sequence and status of entries), and delete specific entries from the queue entry file.

Adding an Entry to a Queue

A client process adds an entry to the specified queue entry file with the AddQueueEntry operation. The client process specifies information, including:

a queue name that must correspond to a queue name contained in the queue index file,

a priority level (0-9 with 0 the highest), at which the entry is queued,

a pointer to a buffer containing the type-specific portion of the queue entry,

an optional time specification for the earliest time the entry is serviced, and

an optional time interval for requeueing of the entry after its removal from the queue entry file. The time interval is added to the time specification for servicing the entry.

Before adding a new entry to the queue entry file, the queue manager checks the number of active server processes. If no server processes are actively serving the queue, some client processes may not wish to queue a new entry.

Reading Queue Entries

A client process reads queue entries with the ReadNextQueueEntry operation for each entry to be read. ReadNextQueueEntry is typically used to list the contents of all entries by utilities such as the Spooler utility (see the "Printer Spooler Utilities Overview" in the B20 System Software Operation Guide, form 1148772).

Queue Entry Handle

The client process specifies the queue name, queue entry handle (see below), and pointers to buffers to which the queue entry and Queue Status Block (see below) are returned.

A queue entry handle is a 32-bit integer that uniquely identifies a queue entry. The control portion of the queue entry (the first 40 bytes that are reserved for the queue manager) contains the queue entry handle of the logically following queue entry.

Queue Status Block

The MarkKeyedQueueEntry, MarkNextQueueEntry (see "Marking Queue Entries" below), and ReadQueueEntry operations take a parameter that is the memory address of a Queue Status Block. These operations use the Queue Status Block to report a queue entry's server user number, priority, and the buffers in which the queue entry handles for the queue entry and the logically following queue entry are stored.

The format of the Queue Status Block is shown in Table 15-2 below. The Queue Status Block is part of the control portion of the queue entry (the first 40 bytes that are reserved for the queue manager).

Table 15-2. Queue Status Block

Offset	Field	Size (Bytes)
0	qehRet	4
4	priority	1
5	ServerUserNum	2
7	qehNextRet	4

where

qehRet   is the buffer in which the queue entry handle of the queue entry is stored.

priority   is the priority (0-9, with 0 the highest) at which the queue entry is placed in the queue.

serverUserNum

        is a 16-bit user number that uniquely identifies the server in the master workstation. If a server marks the entry 0FFFFh, the entry is unmarked.

qehNextRet   is the buffer in which the queue entry handle of the logically following queue entry is stored.

Removing an Entry

A client process removes a specific queue entry from the queue with the RemoveKeyedQueueEntry operation. The queue entry is identified by one or two key fields.

A key is a particular field or combination of fields in a data record upon which the lookup process is performed. The RemoveKeyedQueueEntry operation can specify that up to two key fields must match corresponding fields in the queue entry before the queue entry is removed.

SERVER OPERATIONS

A server process can:

establish itself as an active server for the specified queue(s),

mark and obtain queue entries for processing, and

unmark queue entries or remove itself as an active server.

Establishing Servers

A server process must establish itself as a server for a specific queue entry file with the EstablishQueueServer operation before it can service the queue.

EstablishQueueServer enables the queue manager to keep a count of the number of servers servicing each queue entry file. The queue manager checks the count of servers before adding entries to a queue entry file. If no servers are active, a client process may not wish to queue a new entry.

Marking Queue Entries

The server process obtains a queue entry on which to operate with either of two operations:

the MarkNextQueueEntry operation to specify the next available queue entry, or

the MarkKeyedQueueEntry operation to specify a specific queue entry.

The queue manager marks the specified queue entry as being in use to prevent other server processes from operating on it.

The marking operations prevents interference among multiple server processes servicing a single queue entry file. When a queue entry is marked, it is not returned in subsequent marking operations.

Unmarking Queue Entries

Entries are reset to the unmarked (not in use) state when:

the queue manager is installed,

a server process terminates operation for any reason, including malfunction of a cluster workstation. The queue manager searches all queue entry files affected and resets any queue entries marked by server processes from the malfunctioning workstation.

a server process no longer wishes to service a queue entry file and issues a TerminateQueueServer operation. The queue manager decrements the count of active servers for that queue and resets all entries previously marked by the terminating server.

Sample Queue Entry

Table 15-3 below shows a sample queue entry for printer spooler scheduling. The queue entry format can also be used for user-defined server processes. Queue entries must be large enough to accommodate the control portion of the queue entry (40 bytes that are reserved by the queue manager).

Table 15-3. Sample Queue Entry

Offset	Field	Size (bytes)	Description
0	fDelAftPrt	1	If set to TRUE (0FFh) the spooled file will be deleted after it is printed.
1	sbFileSpec	92	The name of the file to be printed (the first byte of an "sb" string is the length of the string).
93	sbFormName	13	The name of the forms to be used. If the length is zero, the standard form will be used.
106	sbWheelName	13	The name of the print wheel to be used. If the length is zero, the standard print wheel will be used.
119	cCopies	2	The number of copies of the file that are to be printed.
121	bPrintMode	1	The printing mode, where 0 means normal, 1 means binary, and 2 means image mode.
122	fAlignForms	1	If this is TRUE, the forms alignment option will be used.
123	fSecurityMode	1	If this is TRUE, the file will be printed in security mode.
124	reserved	5	Reserved for use by the WRITEone.

Offset	Field	Size (bytes)	Description
129	sbDocName	92	The name of the document being printed. This is different from sbFileSpec, which is typically a temporary file in the [!Scr]<Spl> directory.
221	sbUserName	31	The client's user name.
232	reserved	2	Reserved for use by the WRITEone.
234	reserved	2	
236	timeQueued	4	The date/time that the print was queued.
240	fSupressNewPage	1	If TRUE, the Spooler Manager will not print a form-feed at the start of the print.
241	fWPPaging	1	If TRUE, the Spooler Manager will use WP page escape sequences to determine page numbers.
242	fSupressBanner	1	If TRUE, the Spooler Manager will not print a banner or the notice file.
243	fSingleSheet	1	If TRUE, the printer attached is manual feed.
244	reserved	20	Reserved for future expansion.

Control Queues

A control queue is required for each printer in the system. Control queues store the printer control requests required by the printer spooler for processing. When the user invokes the Spooler utility and uses its printer control subcommands, entries are automatically placed in the associated control queue entry file. The printer spooler queries the control queue entry file periodically to check for printer control entries. The format of the control queue entry is shown below.

Printer Spooler Control Queue Entry Format

Offset	Field	Size (bytes)	Description
0	bCommand	1	The command to the spooler, where 0 means halt/pause printer, 1 means cancel print, 2 means restart printer, and 3 means align forms.
1	restartPage	2	The page number from which to restart printing. If this value is 0 then the printing restarts at the beginning of the current page. If this value is 0FFFFh then the printing starts at the next character in the file.
3	sbWpRestartPage	13	Character sequence that defines a page number in a WRITEone print file.

SpoolerStatus Queue

A single, system-wide status queue named SpoolerStatus stores information about each spooled printer in the cluster configuration. Each printer spooler periodically updates the entry for each printer under its control and for special events. The status information contained in the SpoolerStatus queue is accessed when the user invokes the Spooler utility. The format of the status queue is shown below.

If an application system is reading through a status queue with the ReadNextQueueEntry operation, the only active (valid) printer status entries are those that are marked. An application system can determine which entries are marked by looking at the serverUserNum field of the returned Queue Status Block. If the serverUserNum field is 0FFFFFh, the entry is not an active printer.

Printer Spooler Status Queue Entry Format

Offset	Field	Size (bytes)	Description
0	sbPrinterName	13	The name of the printer
13	sbCurrentPage	13	Character sequence that defines a page number in a WRITEone print file.
26	reserved	25	
51	sbQueueName	51	The name of the queue the printer is serving
102	bChannelNum	1	The channel used, where
			0 (30h) means parallel port,
			A (41h) means serial channel A, and
			B (42h) means serial channel B
103	sbCongfigFile	79	The name of the printer configuration file.

15-17

Offset	Field	Size (bytes)	Description
182	fAtMaster	1	TRUE if server is located at the Master of a cluster.
183	bStatus	1	The printer status, where status means: 0 idle 1 paused 2 printing 3 offline 4 down
184	sbSpooledFile	79	The name of the currently printing file.
263	sbWheelName	13	The name of the current print wheel. If the length is zero, the standard print wheel is used.
276	sbFormName	13	The name of the current forms. If the length is zero, then the standard forms are being used.
289	sbPauseMessage	61	The pause message to be displayed.
350	fNeedWheelChange	1	TRUE if a different print wheel is needed.
351	fNeedFormsChange	1	TRUE if a different form is needed.
352	fShowPauseMsg	1	TRUE if the pause message should be displayed.
353	wsNum	2	The workstation number.
355	reserved		
357	sbDocName	79	The name of the document being printed.
436	sbUserName	31	The client's user name.
467	timeStarted	4	The date/time that the print was started.

Printer Spooler Escape Sequences

Printer spooler escape sequences are special character sequences embedded in text files to be printed by the printer spooler. They either cause an intentional manual intervention condition when processed by the printer spooler or override the page count generated by the printer spooler. The format for a printer spooler escape sequence is:

OFFh, type, cbText, text

where

type   indentifies the reason a manual intervention is required:

                        1 = forms change,
                        2 = print wheel change,
                        3 = generic printer pause, or
                        4 = page number overwrite.

cbText   is the count of bytes in the following text. The maximum is 12 for types 1 and 2, and 60 for type 3.

text   is a character string that identifies the desired form or print wheel, the reason for the generic printer pause, or the page number.

OPERATIONS: SERVICES

Queue management operations are categorized by user group.

Client Process Group

AddQueueEntry
>adds an entry to the specified queue entry file for processing by the appropriate queue server.

ReadKeyedQueueEntry
>obtains the first queue entry in the specified queue entry file with up to two key fields equal to the values specified, reads it into a buffer, and returns the Queue Status Block.

ReadNextQueueEntry
>reads an entry from the specified queue entry file into a buffer and returns the queue entry handle of the next queue entry.

RemoveKeyedQueueEntry
>locates an unmarked entry in the specified queue entry file with up to two key fields equal to the values specified and removes it from the queue entry file.

Server Process Group

EstablishQueueServer
>establishes that a server process wishes to service the specified queue entry file.

MarkKeyedQueueEntry
>locates the first unmarked entry in the specified queue entry file with up to two key fields equal to the values specified, marks it into a buffer, and returns a queue entry handle for use in a subsequent RemoveMarkedQueueEntry operation.

MarkNextQueueEntry

reads the first unmarked entry in the specified queue entry file into a buffer, marks it as being in use, and returns a queue entry handle. Entries are marked in order of priority.

RemoveMarkedQueueEntry

removes a previously marked entry from the specified queue entry file.

RewriteMarkedQueueEntry

rewrites the specified marked queue entry with a new queue entry.

TerminateQueueServer

process is no longer servicing the specified queue entry file.

UnmarkQueueEntry

resets the specified queue entry as unmarked (not in use).

AddQueueEntry

The AddQueueEntry service is used by the client process to add a queue entry to the specified queue entry file for processing by the appropriate queue server.

Procedural Interface

```
AddQueueEntry (pbQueueName, cbQueueName,
        fQueueIfNoServer, priority,
        queueType, pEntry, sEntry,
        pDateTime, repeatTime): ErcType
```

where

pbQueueName
cbQueueName

describe a queue name corresponding to a queue name specified in the queue index file.

fQueueIfNoServer
is TRUE or FALSE.

If fQueueIfNoServer is TRUE, the queue manager adds the entry to the specified queue entry file whether or not servers are active.
If fQueueIfNoServer is FALSE, the queue manager returns status code 908 ("Queue not served") when no servers are active for the specified queue entry file.

priority
is the priority (0-9, with 0 the highest) at which the entry is placed in the queue entry file.

queueType
is the type of the queue (an integer less than or equal to 255), which is used in a consistency check. The queue manager checks the type against the type in the queue index file.

pEntry
sEntry

describe the buffer that contains the type-specific portion of the queue entry.

pDateTime   is a pointer to the 32-bit date/time
                in B20 format (described in the
                "Time Management" section). It
                specifies the earliest time the
                queue entry is served. A 0 means
                the entry is served before an entry
                with a time specification.

repeatTime   specifies the repeating time
                interval in minutes (up to 65,335
                minutes) at which the queue entry is
                serviced. A 0 means no repetition
                occurs. (For example, to repeat the
                entry once a day, specify 1,440
                minutes; to repeat the entry once
                each week, specify 10,080 minutes.)

Request Block

sDateTime   is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	137
12	fQueueIfNoServer	1	
13	priority	1	
14	queueType	2	
16	repeatTime	2	
18	pbQueueName	4	
22	cbQueueName	2	
24	pEntry	4	
28	sEntry	2	
30	pDateTime	4	
34	sDateTime	2	4

EstablishQueueServer

Description

The EstablishQueueServer service is used by the server process to notify the queue manager that it wishes to service the specified queue entry file. A server process should issue EstablishQueueServer before any other operation to the queue manager.

Procedural Interface

EstablishQueueServer (pbQueueName, cbQueueName,
                      queueType,
                      fUniqueServer): ErcType

where

pbQueueName
cbQueueName   describe a queue name corresponding to the queue name specified in the queue index file.

queueType   is the type of the queue (an integer less than or equal to 255), which is used in a consistency check. The queue manager checks the type against the type in the queue index file.

fUniqueServer
               is TRUE or FALSE.

               If fUniqueServer is TRUE, the requesting process intends to become the unique server of the specified queue. If servers already exist, the queue manager returns status code 914 ("Queue already served"). If the operation succeeds, it prevents other servers from being established for that queue.

               If fUniqueServer is FALSE, the requesting process does not intend to become the unique server of the specified queue.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	146
12	queueType	2	
14	fUniqueServer	1	
15	reserved	1	
16	pbQueueName	4	
20	cbQueueName	2	

MarkKeyedQueueEntry

Description

The MarkKeyedQueueEntry service is used by the server process to obtain the first unmarked queue entry with up to two key fields equal to the values specified. MarkKeyedQueueEntry marks the queue entry as being in use, reads it into a buffer, and returns a queue entry handle by which the queue entry is identified in a subsequent RemoveMarkedQueueEntry operation.

The byte count of at least one key field (either cbKey1 or cbKey2) must be nonzero. If only one is nonzero, only that key field is used in the search. If both are nonzero, both are used in the search.

Each nonzero key field must match a specified sb string in the queue entry. In an sb string, the first byte contains the byte count of the string in binary.

MarkKeyedQueueEntry (pbQueueName,
                     cbQueueName, pbKey1, cbKey1,
                     oKey1, pbKey2, cbKey2,
                     oKey2, pEntryRet, sEntryRet,
                     pStatusBlock,
                     sStatusBlock): ErcType

where

pbQueueName
cbQueueName

describe a queue name corresponding
to a queue name specified in the
queue index file.

pbKey1
cbKey1

describe a key field to be compared
with an sb string located at an
offset oKey1 in the queue entry.

oKey1   is the offset of the sb string key
                field in the queue entry.   The
                offset starts from byte 40 (the
                first byte of the type-specific
                portion of the queue entry).

pbKey2
cbKey2   describe a second key field to be
                compared with an sb string located
                at an offset oKey2 in the queue
                entry.

oKey2   is the offset of the second sb
                string key field in the queue entry.
                The offset starts from byte 40 (the
                first byte of the type-specific
                portion of the queue entry).

pEntryRet
sEntryRet

describe the buffer into which the
queue entry is read.

pStatusBlock
sStatusBlock

describe the buffer into which the
status block for the queue entry is
returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	3
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	142
12	oKey1	2	
14	oKey2	2	
16	pbQueueName	4	
20	cbQueueName	2	
22	pbKey1	4	
26	cbKey1	2	
28	pbKey2	4	
32	cbKey2	2	
34	pEntryRet	4	
38	sEntryRet	2	
40	pStatusBlock	4	
44	sStatusBlock	2	

MarkNextQueueEntry

Description

The MarkNextQueueEntry service is used by the server process to read the first unmarked entry from the specified queue entry file into a buffer for processing. Entries are marked in order of priority.

MarkNextQueueEntry marks the entry as being in use and returns a queue entry handle by which the entry is identified in a subsequent RemoveMarkedQueueEntry operation.

Procedural Interface

```
MarkNextQueueEntry (pbQueueName, cbQueueName,
                    fReturnIfNoEntries,
                    pEntryRet, sEntryRet,
                    pStatusBlock,
                    sStatusBlock): ErcType
where
```

pbQueueName
cbQueueName   describe a queue name corresponding
               to a queue name specified in the
               queue index file.

fReturnIfNoEntries
               is TRUE or FALSE.

               If fReturnIfNoEntries is TRUE, the
               queue manager returns status code
               903 ("No entries available") unless
               an unmarked entry is queued.

               If fReturnIfNoEntries is FALSE, the
               queue manager responds to
               MarkNextQueueEntry only when an
               unmarked entry is queued.

pEntryRet
sEntryRet   describe the buffer into which the
               queue entry is read.

pStatusBlock
sStatusBlock
               describe the buffer into which the
               status block for the queue entry is
               returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	141
12	fReturnIfNoEntries	1	
13	reserved	1	
14	pbQueueName	4	
18	cbQueueName	2	
20	pEntryRet	4	
24	sEntryRet	2	
26	pStatusBlock	4	
30	sStatusBlock	2	

ReadKeyedQueueEntry

The ReadKeyedQueueEntry service is used by the client process to obtain the first queue entry with up to two key fields equal to the values specified. ReadKeyedQueueEntry reads the entry into a buffer and returns the Queue Status Block.

The byte count of at least one key field (either cbKey1 or cbKey2) must be nonzero. If only one is nonzero, only that key is used in the search. If both are nonzero, both are used in the search.

Each nonzero key field must match a specified sb string in the queue entry. In an sb string, the first byte contains the byte count of the string in binary.

Procedural Interface

```
ReadKeyedQueueEntry (pbQueueName,
                     cbQueueName, pbKey1, cbKey1,
                     oKey1, pbKey2, cbKey2,
                     oKey2, pEntryRet, sEntryRet,
                     pStatusBlock,
                     sStatusBlock): ErcType
where
```

pbQueueName
cbQueueName

    describe a queue name corresponding to a queue name specified in the queue index file.

pbKey1
cbKey1

    describe a key field to be compared with an sb string located at an offset oKey1 in the queue entry.

oKey1

    is the offset of the sb string key field in the queue entry. The offset starts from byte 40 (the first byte of the type-specific portion of the queue entry).

pbKey2
cbKey2

    describe a second key field to be compared with an sb string located at an offset oKey2 in the queue entry.

oKey2   is the offset of the second sb string key field in the queue entry. The offset starts from byte 40 (the first byte of the type-specific portion of the queue entry).

pEntryRet
sEntryRet   describe the buffer into which the queue entry is read.

pStatusBlock
sStatusBlock describe the buffer into which the status block for the queue entry is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	3
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	140
12	oKey1	2	
14	oKey2	2	
16	pbQueueName	4	
20	cbQueueName	2	
22	pbKey1	4	
26	cbKey1	2	
28	pbKey2	4	
32	cbKey2	2	
34	pEntryRet	4	
38	sEntryRet	2	
40	pStatusBlock	4	
44	sStatusBlock	2	

ReadNextQueueEntry

Description

The ReadNextQueueEntry service is used by the client process to obtain a list of queue entries. ReadNextQueueEntry reads an entry from the specified queue entry file into a buffer and returns the Queue Status Block, which contains the queue entry handle of the next entry in the queue. The entry data returned begins with byte 40 (first byte of the type-specific portion) of the first sector of the queue entry.

If another client process removes the next queue entry before it is read, status code 904 ("Entry deleted") is returned on any attempt to read that entry.

Procedural Interface

ReadNextQueueEntry   (pbQueueName, cbQueueName,
                      qeh, pEntryRet, sEntryRet,
                      pStatusBlock,
                      sStatusBlock): ErcType

where

pbQueueName
cbQueueName   describe a queue name corresponding
                 to a queue name specified in the
                 queue index file.

qeh   is the 32-bit queue entry handle
                 returned from a previous
                 MarkKeyedQueueEntry or
                 MarkNextQueueEntry operation. A 0
                 indicates the first entry in the
                 queue.

pEntryRet
sEntryRet   describe the buffer into which the
                 queue entry is read.

pStatusBlock
sStatusBlock describe the buffer into which the
                 status block for the queue entry is
                 returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	140
12	qeh	4	
16	pbQueueName	4	
20	cbQueueName	2	
22	pEntryRet	4	
26	sEntryRet	2	
28	pStatusBlock	4	
32	sStatusBlock	2	

RemoveKeyedQueueEntry

Description

> The RemoveKeyedQueueEntry service is used by the client process to locate an unmarked entry in the specified queue entry file with up to two key fields equal to the values specified and to remove the entry from the queue entry file.
>
> The byte count of at least one key field (either cbKey1 or cbKey2) must be nonzero. If only one is nonzero, only that key field is used in the search. If both are nonzero, both are used in the search.
>
> Each nonzero key field must match a specified sb string in the queue entry. In an sb string, the first byte contains the byte count of the string in binary.

Procedural Interface

> RemoveKeyedQueueEntry (pbQueueName, cbQueueName,
>   pbKey1, cbKey1, oKey1,
>   pbKey2, cbKey2,
>   oKey2): ErcType
>
> where
>
> pbQueueName
> cbQueueName
>
> > describe a queue name corresponding to a queue name specified in the queue index file.
>
> pbKey1
> cbKey1
>
> > describe a key field to be compared with an sb string located at an offset oKey1 in the queue entry.
>
> oKey1
>
> > is the offset of the sb string key field in the queue entry. The offset starts from byte 40 (the first byte of the type-specific portion of the queue entry).
>
> pbKey2
> cbKey2
>
> > describe a second key field to be compared with an sb string located at an offset oKey2 in the queue entry.

oKey2   is the offset of the second sb
                 string key field in the queue entry.
                 The offset starts from byte 40 (the
                 first byte of the type-specific
                 portion of the queue entry).

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	138
12	oKey1	2	
14	oKey2	2	
16	pbQueName	4	
20	cbQueName	2	
22	pbKey1	4	
26	cbKey1	2	
28	pbKey2	4	
32	cbKey2	2	

RemoveMarkedQueueEntry

Description

> The RemoveMarkedQueueEntry service is used by the server process to remove a previously marked entry from the specified queue entry file. The queue entry to be removed is identified by a queue entry handle previously returned from a MarkKeyedQueueEntry or MarkNextQueueEntry operation.

Procedural Interface

> RemoveMarkedQueueEntry (pbQueueName, cbQueueName, qeh): ErcType
>
> where
>
> pbQueueName
> cbQueueName describe a queue name corresponding to a queue name specified in the queue index file.
>
> qeh   is the 32-bit queue entry handle returned from a previous MarkKeyedQueueEntry or MarkNextQueueEntry operation.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	143
12	qeh	4	
16	pbQueueName	4	
20	cbQueueName	2	

RewriteMarkedQueueEntry

Description

> The RewriteMarkedQueueEntry service is used by the server process to rewrite the specified queue entry with a new entry. RewriteMarkedQueueEntry can be used to update a field contained in a queue entry. The entry to be overwritten is identified by a queue entry handle returned from a previous MarkKeyedQueueEntry or MarkNextQueueEntry operation.

Procedural Interface

> RewriteMarkedQueueEntry (pbQueueName,
> cbQueueName, qeh,
> pEntry, sEntry): ErcType
>
> where

> pbQueueName
> cbQueueName
>> describe a queue name corresponding to a queue name specified in the queue index file.

> qeh
>> is the 32-bit queue entry handle returned from a previous MarkKeyedQueueEntry or MarkNextQueueEntry operation.

> pEntry
> sEntry
>> describe the buffer into which the type-specific portion of the queue entry is read.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	145
12	qeh	4	
16	pbQueueName	4	
20	cbQueueName	2	
22	pEntry	4	
26	sEntry	2	

TerminateQueueServer

Description

>The TerminateQueueServer service is used by the server process to notify the queue manager that the server process is no longer servicing the specified queue entry file. The server process should use TerminateQueueServer when it terminates under normal circumstances.
>
>TerminateQueueServer unmarks any queue entries that were marked by the terminating server process.

Procedural Interface

>TerminateQueueServer (pbQueueName,
>   cbQueueName): ErcType

where

pbQueueName

cbQueueName   describe a queue name corresponding to a queue name specified in the queue index file.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	147
12	pbQueueName	4	
16	cbQueueName	2	

UnmarkQueueEntry

Description

> The UnmarkQueueEntry service is used by the server process to reset the specified queue entry as being unmarked (not in use). The queue entry to be unmarked is identified by a queue entry handle returned from a previous MarkKeyedQueueEntry or MarkNextQueueEntry operation.

Procedural Interface

> UnmarkQueueEntry (pbQueueName, cbQueueName,
>   qeh):   ErcType
>
> where
>
> pbQueueName
> cbQueueName
>   describe a queue name corresponding to a queue name specified in the queue index file.
>
> qeh   is the 32-bit queue entry handle returned from a previous MarkKeyedQueueEntry or MarkNextQueueEntry operation.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	144
12	qeh	4	
16	pbQueueName	4	
20	cbQueueName	2	

SECTION 16
FILE ACCESS METHODS

OVERVIEW

The file management system provides access to disk file data as randomly addressable 512-byte sectors. Up to 127 sectors can be read or written in a single request. Data is transferred directly between disk and the buffer specified in the read/write request (that is, it is not buffered by the file system). Asynchronous operation (concurrent input/output and computation on behalf of the same process) is a standard feature of the file management system.

Several file access methods (described in detail elsewhere in this Manual) augment the capabilities of the file management system. The file access methods are object module procedures that are located in the standard OS library and linked to application systems as required. They provide buffering and use the asynchronous input/output capabilities of the file management system to automatically overlap input/output and computation.

In contrast to the file management system, which organizes disk file data as unstructured 512-byte sectors, the file access methods organize disk file data as either:

o   an unstructured sequence of bytes,

o   a sequence of variable-length records, or

o   a sequence of fixed-length records.

Whenever a file is organized as a sequence of records, these records are both blocked (that is, as many records as possible are packed in each physical sector) and spanned (that is, logical records are permitted to cross physical sector boundaries).

Generally, a file that is an unstructured sequence of bytes is created and subsequently accessed with the Sequential Access Method. A file that is a sequence of variable-length records is created and accessed with the Record Sequential Access Method. A file that is a sequence of fixed-length records is created and accessed with the Direct Access Method or the Indexed Sequential Access Method.

Characteristics of the File Access Methods

The access methods and their general characteristics are:

Sequential Access Method (SAM) provides primarily sequential, overlapped input and output. (Overlapped means that although the application system makes a call to a SAM operation and that operation returns, input/output can continue overlapped automatically with the computations of the application system.) A SAM file is accessed as an unstructured sequence of bytes. Files can be opened for read, write (which replaces any prior file content), and append. In addition to pure sequential access, there are operations for randomly seeking to a specified logical file address.

Record Sequential Access Method (RSAM) provides sequential, overlapped input and output. An RSAM file is accessed as a sequence of fixed- or variable-length records. Files can be opened for read, write (which replaces any prior file content), and append. In addition to pure sequential access, there are operations for scanning forward to the next well-formed record following detection of a malformed record.

Direct Access Method (DAM) provides, random, nonoverlapped input and output. (Nonoverlapped means that a call to a DAM operation does not return to the application system until all associated input or output is complete.) A DAM file is accessed as a sequence of numbered, fixed-length records. Random access is by record number; the implementation is such that reading or writing records with sequential record numbers provides good sequential performance. Files can be opened for read or modify (permitting selective modification of prior file content).

Indexed Sequential Access Method (ISAM) provides random and sequential, nonoverlapped input and output. ISAM is a multikey, multiuser access method. Each ISAM data set holds one type of data record. All data records in a given ISAM data set have the same size. The size of the data records, the number of keys, and the type of

each key are specified when an ISAM data set is created. An ISAM data set consists of two files: an index file and a data store file.

Hybrid Patterns of Access

In the following sections, a file is often referred to as a SAM file, meaning that access to the file is primarily by means of the Sequential Access Method. The terms RSAM file, DAM file, or ISAM data store file are used similarly.

This usage, while convenient, is oversimplified: any file created with RSAM, DAM, or ISAM can be physically viewed as unstructured and accessed using SAM. Similarly, any file of records created with DAM or ISAM can be physically accessed using RSAM (that is, treating fixed-length records as a special case of variable-length records). Finally, an ISAM data store file contains fixed-length records and therefore can be accessed using DAM.

Although all these hybrid patterns of access are possible, they are not all advisable. For example, reading a DAM file with SAM fetches control bytes along with the DAM record bytes; interpreting these requires special knowledge.

As a second example, an ISAM data store file has an associated index file that must be updated in a complex way when the data store file is modified. If the data store file is modified using ISAM, this is done automatically. If the data store file is updated otherwise, the integrity of the ISAM data set can easily be destroyed.

The hybrid patterns of access below are both useful and safe:

o   Use of RSAM to read, write, or append to a DAM-created file. (However, if, following a write or append to such a file, there are records of different lengths, then the file is subsequently accessible only with RSAM, not with DAM.)

o   Use of DAM to read or modify an RSAM-created file in which all records have the same length.

o   Use of RSAM or DAM to read an ISAM-created file as though it were an unkeyed DAM file, that is, with the records accessed according to their physical ordering.

Modifying and Reading Data Files

The Maintain File utility can modify and/or read RSAM and DAM data files. Maintain File can:

o   verify the file structure,

o   remove malformed record,

o   remove deleted records, and

o   optionally write a log of the verification of the file structure to a file.   (The log always appears on the video display.)

Maintain File is also used with the ISAM Reorganize utility. See the B 20 System ISAM Reference Manual, form 1148723 for more information.

Maintain File is described in the B 20 System Software Operation Guide, form 1148772.

CONCEPTS

RSAM and DAM files and ISAM data store files contain standard record headers, record trailers, and file headers.

Standard Record Header

The offsets described in Table 16-1 below are relative to the beginning of a physical record. A physical record consists of the record header, the record data, and the record trailer stored in contiguous bytes.

Table 16-1. Format of a Standard Record Header

Offset	Field	Size (bytes)	Description
0	qURI	4	Universal Record Identifier. lfa of the physical record.
4	sPhyRecord	2	Size of the physical record, including the record header and trailer.
6	bCheck	1	Status byte that has the following meaning: 0 　Record does not logically exist. 1 　Record previously existed but was deleted. 2-15 　Reserved. 16-255 　Record logically exists. bCheck is set to 16 the first time it is written and is incremented on each subsequent write. bCheck recycles to 16 on overflow.

Standard Record Trailer

The offsets described in Table 16-2 below are
relative to the beginning of a physical record.
sRecord is the logical record size.

Table 16-2. Format of a Standard Record Trailer

Offset	Field	Size (bytes)	Description
sRecord+7	bDoubleCheck	1	Copy of bCheck. If bCheck and bDoubleCheck are not equal, the record is malformed.

Standard File Header

The offsets described in Table 16-3 below are
relative to the beginning of the file. A
standard file header occupies an integral number
of sectors at the start of the file. The header
consists of information common to all standard
access methods, followed by information unique to
the particular access method. If no access-
method-dependent information is present, the
first physical record is located at the beginning
of the second file sector.

Table 16-3. Format of a Standard File Header

Offset	Field	Size (bytes)	Description
0	recordHeader	7	Standard record header with the following contents:
			qURI 0
			sPhyRecord 50
			bCheck 16*
7	rgbSignature	2	Standard file header signature, which is the ASCII characters "am".
9	bFileType	1	Indication of the file type. The type corresponds to the last access method that modified the file.

Value   File Type

    2   RSAM
    4   DAM
    8   ISAM data
          store
    9   ISAM index

All other values
are reserved.

Table 16-3. Format of a Standard File Header (Cont.)

Offset	Field	Size (bytes)	Description
10	sPhyRecordMin	2	
	sPhyRecordMax	2	The minimum and maximum physical record sizes (including the 8-byte standard record header and trailer in the file). DAM can be used to access a file only if sPhyRecordMin is equal to sPhyRecordMax which is equal to or greater than 8.
14	reserved	31	Reserved.
45	cbAmDependent	2	Size of the access-method-dependent information that follows the standard file header.
47	wChecksum	2	A word checksum of the preceding 40 bytes. The standard record header at the beginning of the file header is not included in the checksum.
49	recordTrailer	1	A standard record trailer.

*This value changes as per the description in Table 16-1 above.

OPERATIONS: PROCEDURES

The File Access Methods provide the operation listed below.

GetStamFileHeader
> copies the file header of an RSAM, DAM, or ISAM file into the specified area.

GetStamFileHeader

Description

The GetStamFileHeader procedure copies the file header of an RSAM, DAM, or ISAM file into the specified area. The format of the standard file header is described in the section of that name above.

Procedural Interface

GetStamFileHeader (pbFileSpec, cbFileSpec,
                   pbPassword, cbPassword,
                   pFileHeaderRet): ErcType

where

pbFileSpec
cbFileSpec   describe a character string specifying the name of the file whose header is to be read.

pbPassword
cbPassword   describe a character string specifying the password authorizing the requested file access.

pFileHeaderRet
> describes the memory area into which the file header is copied. The memory area must be at least 512 bytes long and word aligned.

Request Block

GetStamFileHeader is an object module procedure.

SECTION 17

SEQUENTIAL ACCESS METHOD

OVERVIEW

The Sequential Access Method (SAM) provides device-independent access to real devices (such as the video display, printer, files, and keyboard) by emulating a conceptual, sequential, character-oriented device.

SAM augments the device-dependent OS operations that are specific to each kind of peripheral device available on a workstation. These operations maximize efficiency and provide access to all features of the peripheral device hardware. However, in many cases, maximum efficiency and specialized features are not as important as device independence.

Consider a program such as a compiler. It is advantageous if a compiler can accept its source data from either the keyboard or a file and can direct its listing to the video display, a printer, or a file.

Programs of this type characterize their input and output as sequences of characters. Selection at execution time of input/output devices can be accomplished for such programs by allowing each type of real device or file to emulate a conceptual device that accepts or supplies any number of characters, but only in sequence.

To retain a large degree of device independence and yet allow access to a few of the most important device-dependent features, extensions to SAM provide device-dependent operations.

For some application systems, not all of the devices supported by SAM are used. For example, an application system can use SAM only to obtain keyboard input and to display text on the video display. It is possible to customize the selection of device-dependent SAM object modules that are linked with an application system. This customization, SAMGen, is described in the B20 Systems Programmers and Assembler Reference Manual (Part 1), form 1148699.

CONCEPTS

Byte Streams

The Sequential Access Method (SAM) uses real devices such as the video display, printer, files, and keyboard to emulate a conceptual, sequential, character-oriented device known as a byte stream.

A byte stream is a readable (input) or writable (output) sequence of 8-bit bytes. An input byte stream can be read until either the reader chooses to stop reading or until it receives status code 1 ("End of file"). An output byte stream can be written until the writer chooses to stop writing. (Of course there are physical limitations: a file could expand to fill all available disk storage, for example.)

A Byte Stream Work Area (BSWA) is a 130-byte memory work area for the exclusive use of SAM procedures. Any number of byte streams can be open concurrently, using separate BSWAs.

SAM consists of object module procedures contained in the standard OS library.

Using a Byte Stream

The first step in using a byte stream is to call the OpenByteStream operation. The user supplies the specification of the device/file, a password if appropriate, the mode (indicating whether input or output is desired), and the address of the 130-byte BSWA. When calling other operations such as ReadBsRecord, WriteBsRecord, and CloseByteStream, the user supplies the address of the same BSWA.

Predefined Byte Streams for Video and Keyboard

There are two predefined and already opened Byte Stream Work Areas (bsVid for video frame 0 and bsKbd for the keyboard). These special BSWAs are defined in SAM standard object modules. Because these BSWAs are already opened, it is not necessary (nor allowed) to specify them as arguments to OpenByteStream or CloseByteStream. In secondary application partitions, these special BSWAs access System Output (SysOut) and System Input (SysIn) facilities.

Device/File Specifications

The device/file specification in the OpenByteStream operation is any of the following:

{node}[volname]<dirname>filename
File identified by its full file specification. Abbreviated specifications are also allowed. See the "File Management" section.

[Lpt]&[volname]<dirname>filename
Centronics-compatible printer connected to the parallel printer port. See the "Printer Spooler Management" section.

&[volname]<dirname>filename describes an optional configuration file containing the printer characteristics. A default configuration file is used if none is specified. See the Create Configuration File Utility in the B20 System Software Operation Guide, form 1148772, for details about the configuration file.

[Ptr] n&[volname]<dirname>filename
RS-232C-compatible printer. n is A or B to indicate the SIO communications channel to which the printer is connected. See the "Printer Spooler Utilities Overview" in the B20 System Software Operation Guide, form 1148772.

&[volname]<dirname>filename describes an optional configuration file containing the printer characteristics. A default configuration file is used if none is specified. See the Create Configuration File Utility in the B20 System Software Operation Guide, form 1148772, for details about the configuration file.

{node}[queuename]reportname
Spooled Printer.

The queue name is the name of the scheduling queue associated with the

printer spooler. [Spl] is the
default name of the first spooled
printer.

The report name is a text string of
up to 12 characters that is included
in the Spooler utility's status
display.

See the "Printer Spooler Utilities
Overview" in the B20 System Software
Operation Guide, form 1148772.

[Kbd]   Keyboard. This also includes the
                 System Input (SysIn) facility used
                 for submit files and batch jobs.
                 See the "Keyboard Management"
                 section in this Manual.

[Comm] n&[volname]<dirname>filename
                 Communications Channel n (A or B) of
                 the SIO communications controller.

                 &[volname]<dirname>filename de-
                 scribes an optional configuration
                 file containing the communications
                 characteristics. A default
                 configuration file is used if none
                 is specified. See the Create
                 Configuration File Utility in the
                 B20 System Software Operation Guide,
                 form 1148772, for details about the
                 configuration file.

[X25] n&[volname]<dirname>filename
                 X.25 virtual circuit. n is a
                 network identification which
                 currently must be zero.

                 &[volname]<dirname>filename de-
                 scribes an optional configuration
                 file containing the circuit
                 characteristics.

[Nul]   Null device. Input operations
                 always return status code 1 ("End of
                 file"). Output operations discard
                 all output but return status code 0
                 ("OK").

[Vid]   Video frame 0. The frame must be
                 established in advance using the
                 Video Display Management or the B20

Executive. (See the "Video Display Management" section.) In secondary application partitions, [Vid] refers to the System Output (SysOut) facility.

[Vid]n   Video frame <u>n</u>.

Customizing SAM

SAM provides device-independent access to a variety of devices. The code that implements SAM is divided into a device-independent portion and several device-dependent portions, one for each kind of device that is supported.

The default SAM supports these devices:

    disk,

    parallel printer,

    spooled printer,

    keyboard,

    null, and

    video display.

For further flexibility, SAM can be customized by a SAM generation (SAMGen) that is similar to a system build. This allows:

reduction of the memory needed by an application system by eliminating unneeded device support,

inclusion of support for communications and serial (RS-232C) printers, and

inclusion of user-written device-specific SAM object modules.

See the B20 System Programmers and Assembler Reference Manual (Part 1), form 1148699 for information on SAMGen.

File Byte Streams

A file byte stream is a byte stream that uses a file on disk. The standard operations of SAM are augmented by two operations that allow random access to files: GetBsLfa and SetBsLfa. These

device-dependent operations are only available for file byte streams and return status code 7 ("Not implemented") if attempted on other byte streams.

Printer Byte Streams

A printer byte stream is a byte stream that performs direct printing. Direct printing transfers text directly from application system memory to the specified parallel or serial printer interface of the workstation on which the application system is executing. A printer byte stream cannot be used to access a printer assigned to the printer spooler.

The selected configuration file determines the printer characteristics. (See the Create Configuration File utility in the B20 System Software Operation Guide, form 1148772.) For example, the configuration file controls whether a printer byte stream suspends execution of the client process until the workstation operator corrects a condition requiring manual intervention or reports it to the calling process.

Normally printer byte streams change tab and end-of-line characters to the form expected by the printer. For example, B20 RETURNs (code 0Ah) can be transformed to carriage return/linefeed combinations for some printers, or just to carriage returns (code 0Dh) or linefeeds (code 0Ah) for others. Tab characters can be transformed to spaces for printers without mechanical tabs. These transformations are controlled by the selected configuration file.

Printing Modes

Any of three printing modes can be specified with the SetImageMode operation: normal, image, or binary. SetImageMode sets the printing mode any time following the opening of the printer byte stream. This differs from the effect of SetImageMode when used with spooler byte streams (see below).

For compatibility between spooled and direct printing, SetImageMode should be used before the first WriteBsRecord or WriteByte operation.

Normal mode converts tabs into spaces and converts end-of-line characters to device-dependent codes.

17-6

<u>Image</u> <u>mode</u> and <u>binary</u> <u>mode</u> perform no code conversion.

Binary mode does not print the banner page, send any extra code not in the file to the printer, nor does it recognize the escape sequences.

Spooler Byte Streams

See the "Printer Spooler Utilities Overview" in the B20 System Software Operation Guide, form 1148772 before using a spooler byte stream and for information on printer spooler escape sequences.

A spooler byte stream automatically creates a uniquely named disk file for temporary text storage. It then transfers the text to the disk file and expands the disk file as necessary. When the spooler byte stream is closed, a request is queued by the printer spooler to the queue manager to print the disk file and delete it after it is printed. This is spooled printing.

Normally, spooler byte streams change tab and end-of-line characters to the form expected by the printer. For example, B20 RETURNs (code 0Ah) can be transformed to carriage return/linefeed combinations for some printers, or just to carriage returns (code 0Dh) or linefeeds (code 0Ah) for others. Tab characters can be transformed to spaces for printers without mechanical tabs. These transformations are controlled by the selected configuration file. (See the Create Configuration File utility in the B20 System Software Operation Guide, form 1148772.)

Printing Modes

Any of three printing modes can be set with the SetImageMode operation: normal, image, or binary. SetImageMode sets the printing mode only if it is called immediately following the opening of the spooler byte stream. This differs from the effect of SetImageMode when used with printer byte streams (see above).

For compatibility between spooled and direct printing, SetImageMode should be used before the first WriteBsRecord or WriteByte operation.

Normal mode prints the banner page between files, converts tabs into spaces, converts end-of-line characters to device-dependent codes, and recognizes the escape sequences for manual intervention. (See the "Printer Spooler Utilities Overview" in the B20 System Software Operation Guide, form 1148772 for information on banner pages.)

Image mode prints the banner page between files and recognizes the escape sequences, but performs no code conversion.

Binary mode does not print the banner, send any extra code not in the file to the printer, nor does it recognize the escape sequences.

Keyboard Byte Streams

A keyboard byte stream is a byte stream that uses the keyboard. The function provided is equivalent to the use of the ReadKbd operation in character mode. (See the "Keyboard Management" section.) The keyboard byte stream does not support unencoded keyboard mode.

To support device-independence, keyboard byte streams return status code 1 ("End of file") when the FINISH (ASCII value 7) key is pressed, and status code 4 ("Operator intervention") when the CANCEL (ASCII value 4) key is pressed.

For applications executing under control of the batch manager in secondary application partitions, keyboard input is received from a System Input (SysIn) facility.

See the "Keyboard Management" section for information on the submit facility and submit file escape sequences.

Communications Byte Streams

A communications byte stream is a byte stream that uses a communications channel. Communications byte streams provide support for the two communications channels of the standard SIO communications controller. Operation is in asynchronous full-duplex mode without explicit modem control. Unlike other byte streams, communications byte streams permit both input and output to be directed to the same open byte stream (that is, the same BSWA). Only one byte

stream can be opened for each communications channel of the SIO.

The selected configuration file determines the communications characteristics. (See the Create Configuration File utility in the B20 System Software Operation Guide, form 1148772.)

Normally, communications byte streams strip null (00h) and delete (7Fh) characters from the stream of received data characters. Image mode (set with the SetImageMode operation) specifies that communications byte streams pass all incoming characters to the client process exactly as received.

X.25 Byte Streams

An X.25 byte stream is a byte stream that enables data transmission via the X.25 Network Gateway.

Each open X.25 byte stream corresponds to a virtual circuit that is initiated when the byte stream is opened, and cleared when the byte stream is closed. Setting up and clearing of the virtual circuit is controlled through the use of a configuration file.

Video Byte Streams

A video byte stream is a byte stream that uses the video display. The standard operations of SAM are augmented by:

Special interpretation of certain characters.

Multibyte escape sequences. The multibyte escape sequences (beginning with the character 0FFh) can be used to control the special video capabilities of the system workstations.

One device-dependent operation. The QueryVidBs operation returns information about video byte streams. (See the description of that operation below.)

For applications executing under control of the batch manager in secondary application partitions, video output is redirected to a System Output (SysOut) facility.

See the "Video Management" section for information on other ways to control the video subsystem.

The characters specially interpreted by video byte streams are described in Table 17-1 below. Note that a multibyte escape sequence (see "Miscellaneous Functions" below) is available that disables all these special interpretations except 0FFh.

Table 17-1. Interpretation of Special Characters

by Video Byte Streams

Hexa-decimal Value	Key	Video Byte Stream Interpretation
01h	up arrow	Move the cursor up one line. If the cursor is in the top line of the frame, reposition it to the bottom line.
07h	CANCEL	Activate audio alarm for one-half second.
08h	BACKSPACE	Backspace one character (with wraparound) and blank that character.
09h	TAB	Tab to next multiple of eight columns. For example, if the cursor is in columns 0-7, it moves to column 8; if it is in columns 8-15, it moves to column 16, etc.
0Ah	RETURN	Move to first column of next line; scroll if necessary.
0Bh	down arrow	Move the cursor down one line. If the cursor is in the bottom line of the frame, reposition it to the top line.
0Ch	NEXT PAGE	Blank the frame and position the cursor in its upper left corner.

Table 17-1. Interpretation of Special Characters

by Video Byte Streams (Cont.)

Hexa-decimal Value	Key	Video Byte Stream Interpretation
0Dh	BOUND	Ignored.
0Eh	left arrow	Move the cursor left one character position. If the cursor is in the first column of the frame, reposition it to the last column.
12h	right arrow	Move the cursor right one character position. If the cursor is in the last column of the frame, re-position it to the first column.
0FFh	CODE-DELETE	Begin multibyte escape sequence.

Multibyte Escape Sequences

Multibyte escape sequences can:

control screen attributes,

control character attributes,

control scrolling and cursor positioning,

dynamically redirect a video byte stream,

automatically pause between full frames of text, and

perform various other miscellaneous functions.

(Note that where the escape sequences include alphabetic characters, upper- and lowercase are equivalent.)

Controlling Screen Attributes. Screen attributes can be controlled with these four multibyte escape sequences. Each of the 3-byte sequences below begins with the escape byte 0FFh and continues with a pair of characters represented by the specified 8-bit ASCII character codes.

Sequence	Effect
0FFh, 'H', 'N'	Turn on the screen half-bright attribute.
0FFh, 'H', 'F'	Turn off the screen half-bright attribute.
0FFh, 'R', 'N'	Turn on the screen reverse video attribute.
0FFh, 'R', 'F'	Turn off the screen reverse video attribute.

Controlling Character Attributes. Character attributes can be controlled with these multibyte escape sequences.

Sequence	Effect
0FFh, 'A', mode	Set the attribute for subsequent characters sent to the frame according to mode, where mode is a single ASCII character defined as follows:

Mode	Blink	Reverse	Underline	Half-bright
'A'	no	no	no	no
'B'	no	no	no	yes
'C'	no	no	yes	no
'D'	no	no	yes	yes
'E'	no	yes	no	no
'F'	no	yes	no	yes
'G'	no	yes	yes	no
'H'	no	yes	yes	yes
'I'	yes	no	no	no
'J'	yes	no	no	yes
'K'	yes	no	yes	no
'L'	yes	no	yes	yes
'M'	yes	yes	no	no
'N'	yes	yes	no	yes
'O'	yes	yes	yes	no
'P'	yes	yes	yes	yes

Sequence	Effect
OFFh, 'A', 'Z'	Enable a mode where writing a character to a character position does not change the character attributes of that character position.

Controlling Scrolling and Cursor Positioning. Characters are normally written to the frame sequentially, with the cursor advancing one character position at a time, from left to right and top to bottom. A cursor is normally displayed at the character position where the next character will be displayed. Text is automatically scrolled each time a character is written to the lower right corner of a frame. When such a scroll occurs, the entire contents of the frame scroll up one line, and the contents of the previous top line of the frame disappear.

The following escape sequences directly control both scrolling and cursor positioning.

Sequence	Effect
OFFh, 'C', x, y	Position the cursor at column x of line y where x and y are single bytes containing (in binary) the column and line numbers. A value of OFFh for x or y specifies, respectively, the last column or line of the frame.
OFFh, 'S', f, l, c, 'D'	Scroll down a portion of the frame. The portion begins at line f(irst) and extends down to, but not including, line l(ast). It is scrolled down by c lines and the top c lines of the scrolled area are filled with the blank character recorded in the bSpace field of the Video Control Block. f, l, and c are single bytes

Sequence	Effect
	containing binary numbers. A value of OFFh for <u>f</u> or <u>l</u> specifies an imaginary line just below the bottom of the frame.
OFFh, 'S', <u>f</u>, <u>l</u>, <u>c</u>, 'U'	Scroll up a portion of the frame.
OFFh, 'V', 'N'	Make the cursor visible.
OFFh, 'V', 'F'	Make the cursor invisible.

Dynamically Redirecting a Video Byte Stream. When a video byte stream is opened, it is designated as directed to one of the frames. However, it is possible to dynamically redirect a video byte stream.

Sequence	Effect
OFFh, 'X', <u>i</u>	Redirect this video byte stream to frame <u>i</u> where <u>i</u> is a single byte containing (in binary) the number of the required frame.

An independent cursor position is recorded for each frame. The position within frame <u>i</u> is restored automatically when a video byte stream is redirected to frame <u>i</u>.

Automatically Pausing Between Full Frames of Text. Automatic pausing between full frames of text can be controlled through multibyte escape sequences. When this pause facility is enabled, and further output to the frame would cause text to be scrolled off the top of the frame, the message:

Press NEXT PAGE to continue

is displayed on the last line of the frame. At this point, if the user presses NEXT PAGE, output continues for another full frame of text. If the user presses CANCEL, status code 4 ("Operator intervention") is returned to the calling process. If the user presses FINISH, status code

1 ("End of file") is returned to the calling process. If the user presses any other key, the audio alarm is momentarily activated.

Since the automatic pause facility reads characters from the keyboard (using the ReadKbdDirect operation; see the "Keyboard Management" section), there is a potential for interaction with the client process's use of the keyboard. A single process that uses a keyboard byte stream and one or more video byte streams will operate correctly.

A more complex environment may require the use of application-specific logic to control pauses in scrolling. Contraindications to automatic pausing are:

> use of the unencoded keyboard mode,
>
> keyboard input performed by a different process than the one using video byte streams, and
>
> keyboard input initiated by the use of the Request primitive and not immediately followed by the Wait primitive.

Sequence	Effect
OFFh, 'P', 'N'	Turn on the pause facility.
OFFh, 'P', 'F'	Turn off the pause facility.

Miscellaneous Functions. The following multibyte escape sequences perform the specified miscellaneous functions.

Sequence	Effect
OFFh, 'L', 'N'	Enable literal mode, with special character interpretation suppressed (except for the escape character OFFh). (See Table 17-1 above.)

17-15

Sequence	Effect
	For example, in literal mode, the character code 08h displays a visible backspace symbol rather than performing the backspace function.
OFFh, 'L', 'F'	Disable literal mode.
OFFh, 'E', 'L'	Erase to the end of the current line of the frame. That is, set the characters to the blank character recorded in the bSpace field of the Video Control Block and turn off all attributes.
OFFh, 'E', 'F'	Erase to the end of the current frame.
OFFh, 'F', <u>char</u>, <u>x</u>, <u>y</u>, <u>w</u>, <u>h</u>	Fill an entire rectangle of the current frame with a single character given by the single byte <u>char</u>. The rectangle is specified by four 1-byte binary numbers: the column and line of the upper left corner (<u>x</u> and <u>y</u>), and the width and height (<u>w</u> and <u>h</u>) of the rectangle. A value of OFFh for <u>x</u> or <u>y</u> specifies, respectively, the last column or line of the frame. A value of OFFh for <u>w</u> or <u>h</u> specifies, respectively, the remaining width or height of the frame.
OFFh, 'I', <u>led</u>, 'N'	Turn on an LED indicator on the keyboard according to the following table:

Sequence	Effect

<table>
<tr><td>LED</td><td>Key</td></tr>
<tr><td>'1'</td><td>f1</td></tr>
<tr><td>'2'</td><td>f2</td></tr>
<tr><td>'3'</td><td>f3</td></tr>
<tr><td>'8'</td><td>f8</td></tr>
<tr><td>'9'</td><td>f9</td></tr>
<tr><td>'0'</td><td>f10</td></tr>
<tr><td>'T'</td><td>OVERTYPE</td></tr>
</table>

OFFh, 'I', <u>led</u>, 'F'   Turn off an LED indicator on the keyboard according to the table immediately above.

OFFh, OFFh   Display a single cross-hatch bar-chart character. The cross-hatch bar-chart character has an 8-bit representation of OFFh (255) and thus cannot be displayed in any other way.

OPERATIONS: PROCEDURES

Sequential Access Method operations are categorized by function in Table 17-2 below.

Table 17-2. Sequential Access Method Operations by Function

Access	Video
CloseByteStream OpenByteStream	QueryVidBs
	File
Input	
	GetBsLfa
ReadBsRecord ReadByte ReadBytes	SetBsLfa
	Other
Output	CheckpointBs PutBackByte
WriteBsRecord WriteByte	ReleaseByteStream SetImageMode

Access

CloseByteStream   closes the open byte stream.

OpenByteStream   opens a device/file as a byte stream.

Input

ReadBsRecord   reads the specified count of bytes from the open input byte stream to the specified memory area.

ReadByte   reads 1 byte from the open input byte stream.

ReadBytes   reads up to the specified count of bytes from the open input byte stream.

Output

WriteBsRecord	writes the specified count of bytes to the open output byte stream from the specified memory area.
WriteByte	writes 1 byte to the open output byte stream.

Video

QueryVidBs	returns information about video byte streams to the client structure.

File

GetBsLfa	returns the logical file address at which the next input/output operation will occur for the open byte stream.
SetBsLfa	sets the logical file address at which the input/output operation is to continue for the open file byte stream.

Other

CheckpointBs	checkpoints the open output byte stream.
PutBackByte	returns 1 byte to the open input byte stream.
ReleaseByteStream	abnormally closes the device/file associated with the open output byte stream.
SetImageMode	sets normal, image, or binary mode.

CheckpointBs

Description

> The CheckpointBs procedure checkpoints the open
> output byte stream identified by the memory
> address of the Byte Stream Work Area.
> CheckpointBs writes any partially full buffers
> and waits for all write operations to complete
> successfully before returning. The byte stream
> remains open for subsequent output.

Procedural Interface

> CheckpointBs (pBSWA): ErcType
>
> where
>
> pBSWA   is the memory address of the same
>   Byte Stream Work Area that was
>   supplied to OpenByteStream.

Request Block

> CheckpointBS is an object module procedure.

CloseByteStream

Description

The CloseByteStream procedure closes the open byte stream identified by the memory address of the Byte Stream Work Area. If the byte stream was open for output, then CloseByteStream writes any partially full buffers and waits for all write operations to complete before returning. After calling CloseByteStream, the process can reuse the Byte Stream Work Area and buffer area. If an error occurs during a CloseByteStream operation, then the byte stream is closed and the error status is returned.

Procedural Interface

CloseByteStream (pBSWA): ErcType

where

pBSWA   is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

Request Block

CloseByteStream is an object module procedure.

GetBsLfa

Description

The GetBsLfa procedure returns the logical file address at which the next input/output operation will occur for the open byte stream identified by the memory address of the Byte Stream Work Area.

GetBsLfa is only valid for file byte streams; otherwise it returns status code 7 ("Not implemented").

Procedural Interface

GetBsLfa (pBSWA, pLfaRet): ErcType

where

pBSWA   is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

pLfaRet   is the memory address of 4 bytes to which the current logical file address is to be returned.

Request Block

GetBsLfa is an object module procedure.

OpenByteStream

Description

The OpenByteStream procedure opens a device/file as a byte stream. If an output byte stream is opened for a file that does not already exist, then OpenByteStream creates it. The address of the Byte Stream Work Area supplied to OpenByteStream must be supplied to subsequent operations such as ReadBytes, WriteBsRecord, and CloseByteStream to identify this particular byte stream.

Procedural Interface

OpenByteStream (pBSWA, pbFileSpec, cbFileSpec,
        pbPassword, cbPassword, mode,
        pBufferArea,
        sBufferArea): ErcType

where

pBSWA   is the memory address of a 130-byte memory work area for use by SAM procedures.

pbFileSpec
cbFileSpec   describe a device or file specification. See the "Device/File Specifications" section above.

pbPassword
cbPassword   describe a device, volume, directory, or file password. The Kbd, Vid, Comm, and Nul devices do not require passwords.

mode   is read, text, write, append, or modify. This is indicated by 16-bit values representing the ASCII constants "mr", "mt", "mw", "ma", or "mm". In these ASCII constants, the first character (m) is the high-order byte and the second character (r, t, w, a, or m, respectively) is the low-order byte.

                 Mode read reads an existing file from the beginning.

Mode text reads an existing file
from the beginning. Mode text is
identical to mode read except when
used to read Word Processor files.
The text of a Word Processor file is
followed by formatting information,
which is not usually desired. When
mode text is specified, status code
1 ("End of file") is returned after
the last byte of text is read (that
is, the formatting information is
ignored).

Mode write overwrites a previously
existing file of the specified name
(if any) and adjusts the length as
necessary. If a file of the
specified name does not exist, SAM
creates one.

Mode append appends output to the
end of an existing file (if any).
If a file of the specified name does
not exist, SAM creates one.

Mode modify is only applicable to
communications byte streams and
indicates that both reading and
writing are allowed on the same
communications channel.

pBufferArea

sBufferArea describe a memory area provided for
the exclusive use of SAM procedures.
To ensure device independence, this
area must be at least 1024 bytes and
word-aligned. Providing a larger
area improves the efficiency of file
operations.

Request Block

OpenByteStream is an object module procedure.

PutBackByte

Description

> The PutBackByte procedure returns 1 byte to the open input byte stream identified by the memory address of the Byte Stream Work Area. This can be useful to a program such as a compiler that may decide, after looking at a character, that it should be processed by a different routine. Only 1 byte can be put back before reading again. An attempt to put back more than 1 byte returns status code 2305 ("Too many put backs").

Procedural Interface

> PutBackByte (pBSWA, b): ErcType
>
> where
>
> pBSWA   is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.
>
> b   is the 8-bit byte to be put back.

Request Block

> PutBackByte is an object module procedure.

QueryVidBs

Description

The QueryVidBs procedure returns information about video byte streams to the client structure.

Procedural Interface

QueryVidBs (pBSWA, pBsVidStateRet): ErcType

where

pBSWA   is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

pBsVidStateRet
                 is the memory address of a 16-byte structure of the format:

Byte	Field	Size (bytes)
0	number of frame	1
1	number of lines in frame	1
2	number of columns in frame	1
3	current line number	1
4	current column number	1
5	cursor visible (TRUE/FALSE)	1
6	pausing between full frames of text enabled (TRUE/FALSE)	1
7	current character attribute mode, as specified in controlling character attributes escape sequence	1
8	literal mode (TRUE/FALSE)	1
9	reserved	7

Request Block

QueryVidBs is an object module procedure.

ReadBsRecord

Description

The ReadBsRecord procedure reads the specified count of bytes from the open input byte stream identified by the memory address of the Byte Stream Work Area to the specified memory area. ReadBsRecord always reads the count of bytes specified except when fewer than that count remain in the file or when an input/output error occurs. If fewer than the specified count of bytes (or no bytes) remain in the file, status code 1 ("End of file") is returned in conjunction with the actual count of bytes read.

Procedural Interface

```
ReadBsRecord (pBSWA, pBufferRet, sBufferMax,
              psDataRet):  ErcType
```

where

pBSWA   is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

pBufferRet   is the memory address of the first byte of the buffer to which the data is to be read.

sBufferMax   is the count of bytes to be read to memory.

psDataRet   is the memory address of the word to which the count of bytes successfully read is returned.

Request Block

ReadBsRecord is an object module procedure.

ReadByte

Description

The ReadByte procedure reads 1 byte from the open input byte stream identified by the memory address of the Byte Stream Work Area. If no bytes remain in the file, status code 1 ("End of file") is returned.

Procedural Interface

ReadByte (pBSWA, pbRet): ErcType

where

pBSWA   is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

pbRet   is the memory address of the byte to which the data is returned.

Request Block

ReadByte is an object module procedure.

ReadBytes

Description

The ReadBytes procedure reads up to the specified count of bytes from the open input byte stream identified by the memory address of the Byte Stream Work Area. The count of bytes made available by this operation is chosen to optimize hardware performance and is not predictable. It can range from 1 to the specified maximum.

ReadBytes returns the memory address of the data bytes in its buffer rather than moving the data to a specified location. This optimizes performance, but imposes the restriction that the calling process must completely process the data before calling ReadBytes again. If this restriction is inconvenient, the ReadBsRecord operation should be used instead. If no bytes remain in the file, status code 1 ("End of file") is returned.

Procedural Interface

ReadBytes (pBSWA, cbMax, ppbRet, pcbRet): ErcType

where

pBSWA   is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

cbMax   is the maximum count of bytes of data that the calling process will accept.

ppbRet   is the memory address of 4 bytes to which the memory address of the data is returned.

pcbRet   is the memory address of a word to which the actual count of data bytes made available is returned.

Request Block

ReadBytes is an object module procedure.

ReleaseByteStream

Description

The ReleaseByteStream procedure abnormally closes the device/file associated with the open output byte stream identified by the memory address of the Byte Stream Work Area. ReleaseByteStream, unlike the CloseByteStream operation, does not properly write remaining partially full buffers. ReleaseByteStream should only be used when a WriteBsRecord, WriteBytes, or CheckpointBs operation fails due to a device error.

Procedural Interface

ReleaseByteStream (pBSWA): ErcType

where

pBSWA   is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

Request Block

ReleaseByteStream is an object module procedure.

SetBsLfa

Description

The SetBsLfa procedure sets the logical file
address at which the input/output operation is to
continue for the open file byte stream identified
by the memory address of the Byte Stream Work
Area. If each of the 4 bytes of the lfa contain
OFFh, then the lfa of the file byte stream is set
to the end-of-file lfa of the file. After
setting the lfa to the end-of-file, the GetBsLfa
operation can be called to determine the length
of the file.

SetBsLfa is only valid for file byte streams;
otherwise, it returns status code 7 ("Not
implemented").

Procedural Interface

SetBsLfa (pBSWA, lfa): ErcType

where

pBSWA   is the memory address of the same
               Byte Stream Work Area that was
               supplied to OpenByteStream.

lfa   is the byte offset, from the
               beginning of the file, of the next
               byte to be read/written.

Request Block

SetBsLfa is an object module procedure.

SetImageMode

Description

The SetImageMode procedure sets the normal, image, or binary mode for printer, spooler, and communications byte streams. SetImageMode, if attempted on other byte streams, returns status code 7 ("Not implemented").

Procedural Interface

SetImageMode (pBSWA, mode): ErcType

where

pBSWA   is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

mode   is a code as follows:

                 0   for normal mode
                 1   for image mode
Request Block   2   for binary mode

SetImageMode is an object module procedure.

WriteBsRecord

Description

The WriteBsRecord procedure writes the specified count of bytes to the open output byte stream identified by the memory address of the Byte Stream Work Area from the specified memory area. Because output is buffered, there is no guarantee of the time at which output is actually written. Only the CheckpointBs and CloseByteStream operations ensure that data was actually written.

Procedural Interface

WriteBsRecord (pBSWA, pb, cb, pcbRet): ErcType

where

pBSWA   is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

pb   is the memory address of the data to be written.

cb   is the count of bytes to write.

pcbRet   is the memory address of the word to which the count of data bytes successfully written is returned.

Request Block

WriteBsRecord is an object module procedure.

WriteByte

Description

The WriteByte procedure writes 1 byte to the open output byte stream identified by the memory address of the Byte Stream Work Area. Because output is buffered, there is no guarantee of the time at which output is actually written. Only the CheckpointBs and CloseByteStream operations ensure that data was actually written.

Procedural Interface

WriteByte (pBSWA, b): ErcType

where

pBSWA   is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

b   is the 8-bit byte to write.

Request Block

WriteByte is an object module procedure.

SECTION 18
RECORD SEQUENTIAL ACCESS METHOD

OVERVIEW

The <u>Record Sequential Access Method</u> (RSAM) provides efficient sequential access to fixed- and variable-length records. Records are read and written using sequential, overlapped input and output. Records are both <u>blocked</u> (that is, as many records as possible are packed in each physical sector) and <u>spanned</u> (that is, logical records are permitted to cross physical sector boundaries). There is also an operation to scan forward to the next well-formed record following detection of a malformed record. Files can be opened for read, write (which replaces any prior file content), and append.

RSAM can be called directly from any of the B20 programming languages. RSAM is a library of object module procedures.

CONCEPTS

RSAM Files and Records

The <u>Record Sequential Access Method</u> (RSAM) provides efficient sequential access to fixed- and variable-length records in a file. An RSAM file is a sequence of these records.

A record can be as large as 65,527 bytes or as small as 1 byte. Records are packed into disk sectors to provide efficient disk storage use. RSAM packs records on write and unpacks them on read. The structure of RSAM records, record headers, record trailers, and of the initial sectors of an RSAM file are explained in the "File Access Methods" section.

If a sector cannot be read or a record is malformed, the remainder of the file can be read after the ScanToGoodRsRecord operation is used to locate the next well-formed record.

Working Area

RSAM uses a work area supplied by the application system. A <u>Record Sequential Work Area</u> (RSWA) is a 150-byte memory work area for the exclusive use of the RSAM procedures. Any number of RSAM files can be open simultaneously using separate RSWAs.

Buffer

RSAM also uses a word-aligned buffer supplied by the application system. The buffer must be at least 1024 bytes long. The buffer size is not constrained by the longest record to be read or written, but performance is improved with the use of large buffers.

RSAM uses overlapped output. Therefore data written to an RSAM file can be retained in the buffer and not actually written to the file until sometime after the WriteRsRecord operation returns. The CheckpointRsFile operation flushes the buffers of an RSAM file, ensuring that all data was written to disk.

18-2

OPERATIONS: PROCEDURES

The Record Sequential Access Method provides the operations listed below.

CheckpointRsFile   checkpoints the open output RSAM file.

CloseRsFile   closes an RSAM file (including conclusion of all input/output operations).

GetRsLfa   returns the logical file address at which the next input/output operation will occur.

OpenRsFile   opens or creates an RSAM file.

ReadRsRecord   reads the next record from an RSAM file.

ReleaseRsFile   releases all resources associated with an open RSAM file (for example, open files and exchanges).

ScanToGoodRsRecord
    scans forward to the next well-formed record in an RSAM file.

WriteRsRecord   writes a record to an RSAM file.

CheckpointRsFile

Description

The CheckpointRsFile procedure checkpoints the open output RSAM file identified by the memory address of the Record Sequential Work Area. CheckpointRsFile writes any partially full buffers and waits for all write operations to complete before returning. The RSAM file remains open for subsequent output.

Procedural Interface

CheckpointRsFile (pRSWA): ErcType

where

pRSWA   is the memory address of the same Record Sequential Work Area that was supplied to OpenRsFile.

Request Block

CheckpointRsFile is an object module procedure.

CloseRsFile

Description

The CloseRsFile procedure closes the open RSAM file identified by the memory address of the Record Sequential Work Area. If the RSAM file was open for output, CloseRsFile writes any partially full buffers and waits for all write operations to complete before returning. After calling ClosRsFile, the Record Sequential Work Area and buffer area can be reused. If an error occurs during a CloseRsFile operation, the RSAM file is closed and the pertinent status code is returned.

Procedural Interface

CloseRsFile (pRSWA): ErcType

where

pRSWA   is the memory address of the same Record Sequential Work Area that was supplied to OpenRsFile.

Request Block

CloseRsFile is an object module procedure.

GetRsLfa

Description

The GetRsLfa procedure returns the logical file address at which the next input/output operation will occur for the open RSAM file identified by the memory address of the Record Sequential Work Area.

Procedural Interface

GetRsLfa (pRSWA, pLfaRet): ErcType

where

pRSWA   is the memory address of the same Record Sequential Work Area that was supplied to OpenRsFile.

pLfaRet   is the memory address of 4 bytes to which the current logical file address is to be returned.

Request Block

GetRsLfa is an object module procedure.

OpenRsFile

Description

The OpenRsFile procedure opens an RSAM file in read, write, or append mode. For write and append modes, if the file does not exist, it is created. The address of the Record Sequential Work Area supplied to OpenRsFile must be supplied to subsequent RSAM operations.

Procedural Interface

OpenRsFile (pRSWARet, pbFilespec, cbFilespec, pbPassword, cbPassword, mode, pBufferArea, sBufferArea): ErcType

pRSWARet   is the memory address of a 150-byte memory work area for use by the Record Sequential Access Method procedures.

pbFilespec
cbFilespec   describe a character string specifying the name of the file to be opened.

pbPassword
cbPassword   describe a character string specifying a password authorizing the requested file access.

mode   is read, write, or append. This is indicated by 16-bit values representing the ASCII constants "mr" (mode read), "mw" (mode write), or "ma" (mode append). In these ASCII constants, the first character (m) is the high-order byte and the second character (r, w, or a, respectively) is the low-order byte.

Mode read reads an existing file from the beginning.

Mode write overwrites a previously existing file of the specified name (if any) and adjusts the length as necessary. If a file of that name does not exist, RSAM creates one.

Mode append appends the records written to the end of an existing file (if any). If a file of the specified name does not exist, RSAM creates one.

pBufferArea
sBufferArea   describe a memory area provided for the exclusive use of the RSAM procedures. This area must be at least 1024 bytes long and word-aligned. Providing a larger area improves the efficiency of RSAM operations.

Request Block

OpenRsFile is an object module procedure.

ReadRsRecord

Description

The ReadRsRecord procedure reads the next record from the open RSAM file identified by the memory address of the Record Sequential Work Area.

Procedural Interface

ReadRsRecord (pRSWA, pRecordRet, sRecordMax, pcbRet): ErcType

where

pRSWA   is the memory address of the same Record Sequential Work Area that was supplied to OpenRsFile.

pRecordRet
sRecordMax   describe the memory area to which the record is to be read.

pcbRet   is the memory address of the word to which the number of bytes read is returned. If the record fits in the supplied memory area, pcbRet is the length of the record. If the record does not fit in the supplied memory area, pcbRet is sRecordMax and status code 3606 ("Record too large") is returned.

Request Block

ReadRsRecord is an object module procedure.

ReleaseRsFile

Description

> The ReleaseRsFile procedure abnormally closes the
> file associated with the open output RSAM file
> identified by the memory address of the Record
> Sequential Work Area. ReleaseRsFile, unlike the
> CloseRsFile operation, does not properly write
> remaining partially full buffers. ReleaseRsFile
> should only be used when a WriteRsRecord of
> CheckpointRsFile operation fails because of a
> device error.

Procedural Interface

> ReleaseRsFile (pRSWA): ErcType
>
> where
>
> pRSWA   is the memory address of the same
>   Record Sequential Work Area that was
>   supplied to OpenRsFile.

Request Block

> ReleaseRsFile is an object module procedure.

ScanToGoodRsRecord

Description

The ScanToGoodRsRecord procedure is called after an attempt is made to read a malformed record or a disk error occurs while reading the open RSAM file identified by the memory address of the Record Sequential Work Area. ScanToGoodRsRecord searches the sectors of the RSAM file until a valid record header is found. The double-check byte of the record found and the header of the following record are then checked, and if they are valid, the RSAM file is positioned to the record found. If the RSAM file is also a Direct Access Method (DAM) file, that is, a file of fixed-length records, record headers are only searched for at the positions where they can occur. These positions are computed by simple arithmetic involving the record length.

ScanToGoodRsRecord reads every sector in the area scanned, so that sectors of the file that were damaged are detected and skipped.

Procedural Interface

ScanToGoodRsRecord (pRSWA, qbSkipMax,
                    pLfaScanStartRet,
                    pqbRet): ErcType

where

pRSWA   is the memory address of the same Record Sequential Work Area that was supplied to OpenRsFile.

qbSkipMax   is a 32-bit unsigned integer (the maximum number of bytes to skip while scanning).

pLfaScanStartRet
                is the memory address of a logical file address to which the byte offset in the RSAM file of the first byte skipped is returned.

pqbRet   is the memory address of a 32-bit unsigned integer to which the number of bytes skipped is returned.

Request Block

ScanToGoodRsRecord is an object module procedure.

WriteRsRecord

Description

The WriteRsRecord procedure writes a record to
the open RSAM file identified by the memory
address of the Record Sequential Work Area. The
RSAM file is automatically extended to
accommodate new records. Because output is
buffered, there is no guarantee of the time at
which output is actually written. Only the
CheckpointRsFile and CloseRsFile operations
ensure that data was actually written.

Procedural Interface

WriteRsRecord (pRSWA, pRecord, sRecord): ErcType

where

pRSWA   is the memory address of the same
               Record Sequential Work Area that was
               supplied to OpenRsFile.

pRecord
sRecord   describe the memory area containing
               the record to be written.

Request Block

WriteRsRecord is an object module procedure.

SECTION 19
DIRECT ACCESS METHOD

OVERVIEW

The <u>Direct Access Method</u> (DAM) provides efficient random access to fixed-length records. A record is referred to in DAM by the record number within a file.

DAM can be accessed in the COBOL language through COBOL Relative I-O. DAM can also be called directly from any of the B20 programming languages. DAM is a library of object module procedures.

In reading, writing, or deleting, DAM does simple address calculations based on the record number to find the required sectors of the DAM file. DAM keeps a cache of recently referenced sectors that are obtained without reference to the disk. Sectors not in the cache are accessed with a single disk access.

CONCEPTS

DAM Files, Records, and Record Fragments

The Direct Access Method (DAM) provides efficient random access to records identified by record number within a file. A record number specifies the record position relative to the first record in a file. The record number of the first record in a file is 1.

A DAM file is a sequence of fixed-length records. The length of a record is specific to each DAM file and is specified when the file is first created.

A record can be as large as 63,992 bytes or as small as 0 bytes. Records are packed into disk sectors to provide efficient disk storage use. DAM packs records on write and unpacks them on read. A packed record contains eight bytes of header and trailer in addition to the stored data.

A record fragment is a contiguous area of memory within a record. A record fragment is specified using an offset from the beginning of the record and a byte count. The record fragment must be contained within the record.

Working Area

DAM uses a work area supplied by the application system. A Direct Access Work Area (DAWA) is a 64-byte memory work area for the exclusive use of the DAM procedures. Any number of DAM files can be open simultaneously using separate DAWAs.

Buffer

DAM also uses a word-aligned buffer supplied by the application system. The buffer size is specified by the application system, subject only to the constraint that it be a multiple of 512 greater than or equal to the record size plus 519. This constraint can be relaxed in two cases. First, if 512 is a multiple of the record size plus eight, the minimal size is simply 512. Second, if the record size plus eight is a multiple of 512, the minimal size is the record size plus eight.

19-2

Buffer Size and Sequential Access

DAM reads and writes the buffer by using a single request to the file management system. This typically requires only a single disk access. Whenever the disk is read, the entire buffer is filled. If the buffer size is chosen to be larger than the record size (by at least a factor of two), the buffer acts as a look-ahead cache: if sequentially numbered records are requested, DAM typically finds them in the buffer and does not access the disk. In this way, if the application system makes a suitable choice of buffer size, the Direct Access Method can provide efficient, record sequential access.

Buffer Management Modes: Write-Through and Write-Behind

DAM provides two modes of buffer management: write-through and write-behind. The mode is initially set to write-through when a DAM file is opened. The mode can be changed using the SetDaBufferMode operation.

In the write-through mode, DAM immediately writes the changed sectors of the buffer to disk whenever a record is written or deleted. DAM guarantees that the file content on disk is accurate at the completion of a modify operation.

In the write-behind mode, DAM writes changed sectors of the buffer to disk only when new sectors are brought into the buffer, the DAM file is closed, or the mode is changed to write-through. Write-behind mode provides better performance when DAM is used to modify records in sequential order.

OPERATIONS: PROCEDURES

Direct Access Method operations are categoried by function in Table 19-1 below.

Table 19-1. Direct Access Method Operations by Function

Access	Input
CloseDaFile	ReadDaFragment
OpenDaFile	ReadDaRecord
Inquiry	**Output**
QueryDaLastRecord	DeleteDaRecord
QueryDaRecordStatus	WriteDaFragment
	WriteDaRecord
Other	
SetDaBufferMode	
TruncateDaFile	

Access

CloseDaFile	closes a DAM file.
OpenDaFile	opens or creates a DAM file.

Input

ReadDaFragment	read a record fragment from an open DAM file.
ReadDaRecord	reads a record from a DAM file.

Output

DeleteDaRecord	deletes a record from a DAM file.
WriteDaFragment	write a record fragment to an open DAM file.
WriteDaRecord	writes a record to a DAM file.

Inquiry

QueryDaLastRecord
> copies to the specified area the number of the last record in an open DAM file.

QueryDaRecordStatus
> copies to the specified area the status of a record in an open DAM file.

Other

SetDaBufferMode
> sets the buffer management mode to write-through or write-behind.

TruncateDaFile
> truncates an open DAM file (that is, it removes all records beyond a specified point).

CloseDaFile

Description

The CloseDaFile procedure closes the open DAM file identified by the memory address of the Direct Access Work Area. After calling CloseDaFile, the application system can reuse the Direct Access Work Area and the buffer area.

Procedural Interface

CloseDaFile (pDAWA): ErcType

where

pDAWA   is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

Request Block

CloseDaFile is an object module procedure.

DeleteDaRecord

Description

The DeleteDaRecord procedure deletes a record from the open DAM file identified by the memory address of the Direct Access Work Area. The deleted record is specified by the record number. Once a record is deleted, it can no longer be read.

Procedural Interface

DeleteDaRecord (pDAWA, qiRecord): ErcType

where

pDAWA   is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

qiRecord   is a 32-bit unsigned integer specifying the number of the record to be deleted.

Request Block

DeleteDaRecord is an object module procedure.

OpenDaFile

Description

The OpenDaFile procedure opens a DAM file in either read (shared) or modify (exclusive) mode. If the file does not exist, it is created. The address of the Direct Access Work Area supplied to OpenDaFile must be supplied to subsequent DAM operations.

Access to a DAM file is most efficient if its sectors are physically contiguous. This contiguity can be increased by preallocating the file. To preallocate the file, follow the call to OpenDaFile that creates the file with a call to WriteDaRecord. This call to WriteDaRecord should specify a value of qiRecord large enough to preallocate the desired file length. This "end record" can then be deleted.

Procedural Interface

OpenDaFile (pDAWARet, pbFilespec, cbFilespec,
            pbPassword, cbPassword, mode,
            pBuffer, sBuffer, sRecord): ErcType

where

pDAWARet   is the memory address of a 64-byte memory work area for use by the Direct Access Method procedures.

pbFilespec
cbFilespec   describe a character string specifying the name of the file to be opened.

pbPassword
cbPassword   describe a character string specifying a password that authorizes the requested file access.

mode   is read (shared) or modify (exclusive). This is indicated by 16-bit values representing the ASCII constants "mr" (mode read) or "mm" (mode modify). In these ASCII constants, the first character (m) is the high-order byte and the second character (r or m, respectively) is the low-order byte.

pBuffer
sBuffer   describe a word-aligned memory area
                 provided for the exclusive use of
                 the   Direct   Access   Method
                 procedures. The size of this area
                 is discussed in the "Buffer" section
                 above.

sRecord   describes the fixed record size for
                 the DAM file.   If the DAM file
                 already exists, sRecord must match
                 the record size specified when the
                 file was created.

Request Block

OpenDaFile is an object module procedure.

QueryDaLastRecord

Description

The QueryDaLastRecord procedure copies to the specified area the number of the last record in the open DAM file. The file is identified by the memory address of the Direct Access Work Area. The last record is the existing record having the largest record number.

If the DAM file contains no records, the last record number is 0.

Procedural Interface

QueryDaLastRecord (pDAWA, pqiRecordRet): ErcType

where

pDAWA   is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

pqiRecordRet   is the memory address of the 32-bit memory area to which the last record number is written.

Request Block

QueryDaLastRecord is an object module procedure.

QueryDaRecordStatus

Description

The QueryDaRecordStatus procedure copies to the specified area the status of a record in the open DAM file. The file is identified by the memory address of the Direct Access Work Area. The record status is interpreted in this way:

ercOK   the record exists.

ercRecordDoesNotExist (code 3302)
                 the record does not exist.

ercRecordBeyondExistingRecords (code 3007)
                 the record does not exist. The record has a larger record number than any existing record.

Caution: The status code value returned by QueryDaRecordStatus is the status of the operation, not the record status. The memory address of the record status is passed as a parameter.

Procedural Interface

QueryDaRecordStatus (pDAWA, qiRecord,
                     pStatusRet): ErcType

where

pDAWA   is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

qiRecord   is a 32-bit unsigned integer specifying the number of the record to query.

pStatusRet   is the memory address of a word to which the record status is written.

Request Block

QueryDaRecordStatus is an object module procedure.

ReadDaFragment

Description

The ReadDaFragment procedure reads a record fragment from the open DAM file identified by the memory address of the Direct Access Work Area. The returned record fragment is specified by the record number, relative offset, and byte count.

Procedural Interface

ReadDaFragment (pDAWA, qiRecord, pFragmentRet,
                rbFragment, cbFragment): ErcType

where

pDAWA   is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

qiRecord   is a 32-bit unsigned integer specifying the number of the record containing the record fragment to be read. qiRecord must correspond to an existing record.

pFragmentRet
                is the memory address of the memory area to which the record fragment is returned.

rbFragment   is the offset from the beginning of the record to the first byte of the record fragment.

cbFragment   is the size of the record fragment.

Request Block

ReadDaFragment is an object module procedure.

ReadDaRecord

Description

The ReadDaRecord procedure reads a record from the open DAM file identified by the memory address of the Direct Access Work Area. The returned record is specified by the record number.

Procedural Interface

ReadDaRecord (pDAWA, qiRecord,
              pRecordRet): ErcType

where

pDAWA   is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

qiRecord   is a 32-bit unsigned integer specifying the number of the record to be read. qiRecord must correspond to an existing record.

pRecordRet is the memory address of the memory area to which the record is returned.

Request Block

ReadDaRecord is an object module procedure.

SetDaBufferMode

Description

The SetDaBufferMode procedure sets the buffer management mode to write-through or write-behind. These two buffering modes are described in the "Concepts" section above.

Procedural Interface

SetDaBufferMode (pDAWA, mode): ErcType

where

pDAWA	is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.
mode	is either the write-through or write-behind buffer management mode. This is indicated by 16-bit values representing the ASCII constants "wt" (write-through) and "wb" (write-behind). In these ASCII constants, the first character (w) is the high-order byte and the second character (t or b, respectively) is the low-order byte.

Request Block

SetDaBufferMode is an object module procedure.

TruncateDaFile

Description

The TruncateDaFile procedure truncates the open DAM file (that is, it removes all records beyond a specified point). All records having record numbers greater than the qiRecord parameter are deleted. If qiRecord is 0, all records in the DAM file are deleted.

Procedural Interface

TruncateDaFile (pDAWA, qiRecord): ErcType

where

pDAWA   is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

qiRecord   is a 32-bit unsigned integer specifying a record number. All records having record numbers greater than qiRecord are deleted.

Request Block

TruncateDaFile is an object module procedure.

WriteDaFragment

Description

The WriteDaFragment procedure writes a record fragment to the open DAM file identified by the memory address of the Direct Access Work Area. The written record fragment is specified by the record number, relative offset, and byte count. The DAM file is automatically extended to accommodate new records.

Procedural Interface

WriteDaFragment (pDAWA, qiRecord, pFragment,
                rbFragment, cbFragment): ErcType

where

pDAWA   is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

qiRecord   is a 32-bit unsigned integer specifying the number of the record containing the record fragment to be written.

pFragment   is the memory address of the memory area from which the record fragment is written.

rbFragment   is the offset from the beginning of the record to the first byte of the record fragment.

cbFragment   is the size of the record fragment.

Request Block

WriteDaFragment is an object module procedure.

WriteDaRecord

Description

The WriteDaRecord procedure writes a record to the open DAM file identified by the memory address of the Direct Access Work Area. The written record is specified by the record number. The DAM file is automatically extended to accommodate new records.

WriteDaRecord can write a record with a record number larger than any existing record number. If this is done, the file is extended and standard record header and trailer formats are written automatically to all added sectors. The time required for the WriteDaRecord operation is proportional to the amount by which the file is extended.

Procedural Interface

WriteDaRecord (pDAWA, qiRecord, pRecord): ErcType

where

pDAWA	is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.
qiRecord	is a 32-bit unsigned integer specifying the number of the record to be written.
pRecord	is the memory address of the memory area from which the record is written.

Request Block

WriteDaRecord is an object module procedure.

SECTION 20
INDEXED SEQUENTIAL ACCESS METHOD

OVERVIEW

The Indexed Sequential Access Method (ISAM) provides efficient, yet flexible, random access to fixed-length records identified by multiple keys stored in disk files.

Each ISAM data set holds one type of data record. The size of the data records, the number of keys, and the type of each key are specified when an ISAM data set is created.

ISAM, a software product, is described in the B20 System ISAM Reference Manual, form 1148723.

CONCEPTS
Key Types

A record can have an unlimited number of keys. Each key is described by its position in the record (offset from the first byte of the record), the key length, and the key type.

There are six key types:

o   byte string (up to 64 bytes),

o   character string (up to 64 bytes),

o   packed decimal (COBOL COMP-3),

o   binary,

o   long real, and

o   short real.

Key type is important because the collating sequence depends on it.

Each key defines an index (that is, an inversion) which is automatically updated when records are stored or modified and which is used as the basis of retrieval. Records can be retrieved in key-order sequence by any key field, starting with any key value.

To increase flexibility, the following parameters can be specified for each key at the time an ISAM data set is created:

o   whether duplicates are allowed,

o   whether the index is to be kept in ascending or descending order, and

o   whether indexing of a record whose key field contains a null value is to be suppressed. (Suppressing the indexing of such fields reduces the size of the index.)

File Types

Each ISAM data set holds one record type but is stored as two physical files: a data store file and an index file. These can be placed on different physical volumes if desired.

The data store file holds the data records. Because all the records in a data set have the same length, physical space management that conserves disk space is simple and efficient: whenever a record is deleted, its space is added to a free list and later reused when a new record is created.

The data store file is a Direct Access Method (DAM) file. See the "Direct Access Method" section.

The index file holds all indexes for all of a data set's keys. Each index is implemented as a B-tree. This implementation technique, sometimes called "block splitting," ensures that data records can be repeatedly added without creating long overflow chains or requiring physical reorganization.

Operations

ISAM supports four principal kinds of operations: storing, reading, modifying, and deleting.

When an application system stores a new record, ISAM automatically indexes the record according to the values in all its key fields.

When an application system reads an existing record, it can retrieve any of the following:

o   all records whose keys have a specific value
     (that is, an <u>exact</u> <u>match</u>),

o   all records whose key values lie in a
     specified range (that is, a <u>range</u> <u>match</u>), or

o   all records in which the initial bytes of a
     byte or character string key match a
     particular value (that is, a <u>prefix</u> <u>match</u>).

An application system can retrieve either the
specified records in order, or a sequence of 4-
byte unique record identifiers in order. If
record identifiers are retrieved, then the
application program can later obtain the
corresponding data records by a special form of
the retrieval operation, without reaccessing the
index.

<u>Modifying</u> an existing record combines storing and
reading. Before a record is modified, it is
automatically removed from each index for which
the key field is being changed, then indexed
under the new key field.

ISAM Organization

The ISAM facility consists of:

o   a multiuser access package,

o   a single-user access package, and

o   utilities.

The multiuser access package and the single-user
access package provide identical procedural
interfaces to the application system.

Multiuser Access Package

The ISAM <u>multiuser</u> <u>access</u> package provides shared
access to ISAM data sets from several cluster
workstations. ISAM must be resident on the
master (or standalone) workstation. ISAM
operations are invoked using the standard OS
request model: either by making an Operating
System request or by invoking a procedure (which
automatically makes the request).

Single-User Access Package

The ISAM single-user access package provides exclusive access to ISAM data sets from a single application partition of the workstation on which the application system runs. ISAM must be linked into the application system and then initialized by invoking an initialization procedure. ISAM operations are invoked by calling ISAM procedures directly. ISAM operations can be invoked by only one process at a time.

Utilities

ISAM includes utilities that are invoked from the Executive. The ISAM Create utility creates an empty ISAM data set. The ISAM utilities ISAM Copy, ISAM Rename, ISAM Delete, and ISAM Set Protection provide capabilities for ISAM data sets similar to those the Executive commands Copy, Rename, Delete, and Set Protection provide for individual files. The ISAM Status utility displays information about an ISAM data set. The ISAM Reorganize utility changes the key fields of a data set, loads data from files, and recovers data from data sets that have become malformed.

SECTION 21
DISK MANAGEMENT

OVERVIEW

Disk management operations provide device-level access to disk devices, in contrast to the file-level access provided by file management operations. Access to a disk device at such a level is necessary in order to read a floppy disk written on a non-B20 system or to format an uninitialized disk.

Device-level access is provided to IBM-compatible, single-sided, 8-inch floppy disks written in either single or double density with sector sizes of 128, 256, 512, or 1024. The sector size and density of a floppy disk, if other than 512-byte double density, must be specified with the SetDevParams operation.

CONCEPTS

Accessing a Disk Device

A device can be accessed by using an OpenFile operation with a device or volume specification. The Read, Write, ReadAsync and CheckRead-Async, WriteAsync and CheckWriteAsync, and CloseFile operations all accept a file handle returned by such an OpenFile operation. (File handles are discussed in detail in the "File Management" section.)

Device-level access to disks bypasses the concurrency control of the file management system. Thus extreme care is required if device-level access is used in a cluster configuration.

Device Specification and Device Password

A disk device is a physical hardware entity. Access to a device requires presentation of a device specification and a password. A device specification can take either of two forms, depending on whether the medium of the disk device contains a valid file system.

If a volume contains a valid file system, the device specification has the form:

{node}[volname]

(In this case, the volume password of the volume must be specified. Volume passwords are described in the "File Management" section.)

However, if the medium does not contain a valid file system (either because the medium was never initialized to contain one or because the file system has become malformed), the device specification has the form:

{node}[devname]

(In this case, the device password of the device must be specified. A device password protects a device. It can have a maximum of 12 characters, consisting of all alphanumeric characters plus the period, ".", and the hyphen, "-".)

A volname (volume name) or a devname (device name) is a string of characters. A volname or devname can have a maximum of 12 characters, consisting of all alphanumeric characters, plus the period, ".", and the hyphen, "-".

OPERATIONS: PROCEDURES AND SERVICES

Disk management operations are categorized by function in Table 21-1 below.

Table 21-1. Disk Management Operations by Function

Access	Input/Output
CloseFile	CheckReadAsync
OpenFile	CheckWriteAsync
	Format
Other	Read
	ReadAsync
DismountVolume	Write
GetVHB	WriteAsync
MountVolume	
QueryDCB	
SetDevParams	

Access

CloseFile   closes an open file handle.

OpenFile   opens a device and returns a file
                handle.

Input/Output

CheckReadAsync
                waits for input completion, checks
                the status code, and obtains the
                byte count of data read after a
                ReadAsync procedure.

CheckWriteAsync
                waits for output completion, checks
                the status code, and obtains the
                byte count of data written after a
                WriteAsync procedure.

Format   initializes the surface of a floppy
                disk or other disk media to
                accommodate fixed-size data
                sectors. Used by the IVolume
                utility.

Read | transfers an integral number of 128-, 256-, 512-, or 1024-byte sectors from disk to memory.

ReadAsync | initiates the transfer of an integral number of 128-, 256-, 512-, or 1024-byte sectors from disk to memory. The CheckReadAsync procedure must be used to check the completion status of the transfer.

Write | transfers an integral number of 128-, 256-, 512-, or 1024-byte sectors from memory to disk.

WriteAsync | initiates the transfer of an integral number of 128-, 256-, 512-, or 1024-byte sectors from memory to disk. The CheckWriteAsync procedure must be used to check the completion status of the transfer.

Other

DismountVolume
dismounts the specified volume.

GetVHB | copies the Volume Home Block of the specified device to the specified memory area.

MountVolume | mounts the volume on the specified disk drive.

QueryDCB | copies the Device Control Block of the specified device to the specified memory area.

SetDevParams
allows the characteristics of the floppy disk controller to be modified to accommodate non-B20 floppy disks.

CheckReadAsync

Description

After calling the ReadAsync procedure to initiate a read, the requesting process continues execution. When the process wants to synchronize with the asynchronous read (that is, wait for its completion), the process does a CheckReadAsync. The CheckReadAsync procedure waits for input completion, checks the status code, and obtains the byte count of data read.

Status code 248 ("Wrong pRq argument") is returned if the pRq argument does not match the one of the preceding ReadAsync procedure.

Procedural Interface

CheckReadAsync (pRq, psDataRet): ErcType

where

pRq   is the same memory address as given in the pRq argument of the ReadAsync procedure.

psDataRet   is the memory address of the word to which the count of bytes successfully read is to be returned.

Request Block

The ReadAsync and CheckReadAsync procedures are procedural interfaces to the Read operation. See the Read operation below.

CheckWriteAsync

Description

After calling the WriteAsync procedure to
initiate a write, the requesting process
continues execution. When the process wants to
synchronize with the asynchronous write (that is,
wait for its completion), the process does a
CheckWriteAsync. The CheckWriteAsync procedure
waits for output completion, checks the status
code, and obtains the byte count of data written.

Status code 248 ("Wrong pRq argument") is
returned if the pRq argument does not match the
one of the preceding WriteAsync procedure.

Procedural Interface

CheckWriteAsync (pRq, psDataRet): ErcType

where

pRq   is the same memory address as given
                in the pRq argument of the
                WriteAsync procedure.

psDataRet   is the memory address of the word to
                which the count of bytes success-
                fully written is to be returned.

Request Block

The WriteAsync and CheckWriteAsync procedures are
procedural interfaces to the Write operation.
See the Write operation below.

CloseFile

Description

The CloseFile service closes an open file.

Procedural Interface

CloseFile (fh): ErcType

where

fh   is the file handle returned from an OpenFile operation.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	10
12	fh	2	

DismountVolume

Description

The DismountVolume service dismounts the specified volume.

Dismounting (and mounting) of volumes is normally controlled by the Automatic Volume Recognition (AVR) capability of the file management system. The Dismount (and Mount) operations are provided for the use of utilities, such as IVolume, that must override AVR. (IVolume is described in the B20 System Software Operation Guide, form 1148772.)

Procedural Interface

DismountVolume (pbVolName, cbVolName, pbPassword, cbPassword): ErcType

where

pbVolName
cbVolName   describe a character string of the form {node}[volname]. Square brackets are optional for the device name. The distinction between uppercase and lowercase is not significant in matching device names.

pbPassword
cbPassword   describe the volume password that authorizes access to the specified volume.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	12
12	reserved	6	
18	pbVolName	4	
22	cbVolName	2	
24	pbPassword	4	
28	cbPassword	2	

Format

Description

The Format service initializes the surface of a floppy disk or other disk media to accommodate fixed-size data sectors. Format is used by the IVolume utility (described in the B20 System Software Operation Guide, form 1148772) and is device-dependent.

Procedural Interface

```
Format (fh, pBuffer, sBuffer, lfa,
        psDataRet): ErcType
```

where

fh   is a file handle returned from an OpenFile operation that specifies a device.

pBuffer   is the memory address of the first byte of control information. The buffer must be word aligned.

sBuffer   is the count of bytes of control information to be transferred. It must be a multiple of 2.

lfa   is the byte offset, from the beginning of the device, of the first sector to be initialized.

psDataRet   is the memory address of the word to which the count of bytes successfully transferred is to be returned.

Request Block

ssDataRet is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	38
12	fh	2	
14	lfa	4	
18	pBuffer	4	
22	sBuffer	2	
24	psDataRet	4	
28	ssDataRet	2	2

GetVHB

Description

The GetVHB service copies the Volume Home Block of the specified device to the specified memory area. If the specified area is not large enough to hold the requested information, the information is truncated.

GetVHB does not require a password. To avoid security violations, 0's are returned in the volPassword field of the Volume Home Block.

The Volume Home Block is described in the "File Management" section.

Procedural Interface

GetVHB (pbDevSpec, cbDevSpec, pVhbRet, sVhbMax): ErcType

where

pbDevSpec
cbDevSpec   describe a character string of the form {node}[devname] or {node}[vol- name]. Square brackets are optional for the device name. The distinction between uppercase and lowercase is not significant in matching device names.

pVhbRet
sVhbMax   describe the memory area.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	15
12	reserved	6	
18	pbDevSpec	4	
22	cbDevSpec	2	
24	pVhbRet	4	
28	sVhbMax	2	

MountVolume

Description

The MountVolume service mounts the volume on the specified disk drive.

Mounting (and dismounting) of volumes is normally controlled by the Automatic Volume Recognition (AVR) capability of the file management system. The Mount (and Dismount) operations are provided for the use of utilities, such as IVolume, that must override AVR. (IVolume is described in the B20 System Software Operation Guide, form 1148772.)

Procedural Interface

MountVolume (pbDevSpec, cbDevSpec, pbDevPassword, cbDevPassword): ErcType

where

pbDevSpec
cbDevSpec   describe a character string of the form {node} [devname]. Square brackets are optional for the device name. The distinction between uppercase and lowercase is not significant in matching device names.

pbPassword
cbPassword   describe the device password that authorizes access to the specified device.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	11
12	reserved	6	
18	pbDevSpec	4	
22	cbDevSpec	2	
24	pbPassword	4	
28	cbPassword	2	

OpenFile

Description

The disk management form of the OpenFile service opens the entire specified volume/device as a file and returns a file handle. The file handle returned by OpenFile is used to refer to the file in subsequent operations such as Read, Write, and CloseFile.

Procedural Interface

OpenFile (pFhRet, pbDevSpec, cbDevSpec,
        pbPassword, cbPassword, mode): ErcType

where

pFhRet   is the memory address of the word to
                  which the file handle is returned.

pbDevSpec
cbDevSpec   describe a character string of the
                  form {node}[devname] or {node}[vol-
                  name]. The distinction between
                  uppercase and lowercase is not
                  significant in matching device
                  names.

pbPassword
cbPassword   describe either the device or volume
                  password that authorizes access to
                  the specified device.

mode   is read or modify. This is
                  indicated by 16-bit values
                  representing the ASCII constants
                  "mr" (mode read) and "mm" (mode
                  modify). In these ASCII constants,
                  the first character (m) is the high-
                  order byte and the second character
                  (r or m, respectively) is the low-
                  order byte.

                  Access in read mode permits the
                  returned file handle to be used as
                  an argument only to the Read,
                  ReadAsync, CheckReadAsync, and
                  CloseFile operations. Access in
                  modify mode, however, also permits
                  the returned file handle to be used

as an argument to the WriteAsync,
CheckWriteAsync, and Write opera-
tions.

There is no limit to the number of
concurrent opens of a disk device in
either read or modify mode.

Request Block

sFhMax is the size of a file handle and is always
2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	4
12	reserved	2	
14	mode	2	
16	reserved	2	
18	pbDevSpec	4	
22	cbDevSpec	2	
24	pbPassword	4	
28	cbPassword	2	
30	pFhRet	4	
34	sFhMax	2	2

QueryDCB

Description

The QueryDCB service copies the Device Control Block of the specified device to the specified memory area. If the specified area is not large enough to hold the requested information, the information is truncated.

QueryDCB does not require a password. To avoid security violations, 0's are returned in the devPassword field of the Device Control Block.

The Device Control Block is described in the "File Management" section.

Procedural Interface

QueryDCB (pbDevSpec, cbDevSpec, pDcbRet,
         sDcbMax): ErcType

where

pbDevSpec
cbDevSpec   describe a character string of the
                form {node}[devname] or {node}[vol-
                name]. Square brackets are optional
                for the device name. The
                distinction between uppercase and
                lowercase is not significant in
                matching device names.

pDcbRet
sDcbMax   describe the memory area.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	124
12	reserved	6	
18	pbDevSpec	4	
22	cbDevSpec	2	
24	pDcbRet	4	
28	sDcbMax	2	

Read

Description

The Read service transfers an integral number of
128-, 256-, 512-, or 1024-byte sectors from disk
to memory. Read returns only when the requested
transfer is complete. The ReadAsync and
CheckReadAsync procedures are used to overlap
computation and input/output transfer.

To accommodate programming languages in which
Read is a reserved word, ReadFile is permitted as
a synonym for the Read service.

Procedural Interface

Read (fh, pBufferRet, sBufferMax, lfa,
    psDataRet): ErcType

where

fh   is a file handle returned from an
                    OpenFile operation. The device can
                    be open in either read or modify
                    mode.

pBufferRet   is the memory address of the first
                    byte of the buffer to which the data
                    is to be read. The buffer must be
                    word aligned.

sBufferMax   is the count of bytes to be read to
                    memory. It must be a multiple of
                    the sector size (128, 256, 512, or
                    1024).

lfa   is the byte offset, from the
                    beginning of the file, of the first
                    byte to be read. It must be a
                    multiple of the sector size (128,
                    256, 512, or 1024).

psDataRet   is the memory address of the word to
                    which the count of bytes success-
                    fully read is to be returned.

21-20

Request Block

ssDataRet is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	35
12	fh	2	
14	lfa	4	
18	pBufferRet	4	
22	sBufferMax	2	
24	psDataRet	4	
28	ssDataRet	2	2

ReadAsync

Description

The ReadAsync procedure initiates the transfer of an integral number of 128-, 256-, 512-, or 1024-byte sectors from disk to memory. The CheckRead-Async procedure must be called to check the completion status of the transfer.

The information returned by Read with its psDataRet argument and ErcType status is obtained by CheckReadAsync.

Procedural Interface

ReadAsync (fh, pBufferRet, sBufferMax, lfa, pRq, exchangeReply): ErcType

where

fh   is a file handle returned from an OpenFile operation. The device can be open in either read or modify mode.

pBufferRet   is the memory address of the first byte of the buffer to which the data is to be read. The buffer must be word aligned.

sBufferMax   is the count of bytes to be read to memory. It must be a multiple of the sector size (128, 256, 512, or 1024).

lfa   is the byte offset, from the beginning of the file, of the first byte to be read. It must be a multiple of the sector size (128, 256, 512, or 1024).

pRq   is the memory address of a 64-byte area to be used as workspace by ReadAsync.

exchangeReply
              is an exchange provided by the client process for the exclusive use of ReadAsync and CheckReadAsync.

Request Block

> The ReadAsync and CheckReadAsync procedures are procedural interfaces to the Read operation. See the Read operation above.

SetDevParams

Description

The SetDevParams service allows the characteristics of the 8-inch floppy disk controller to be modified to accommodate non-B20 floppy disks.

Procedural Interface

SetDevParams (pbDevSpec, cbDevSpec, pbPassword, cbPassword, paramCode): ErcType

where

pbDevSpec
cbDevSpec   describe a character string of the form {node}[devname]. Square brackets are optional for the device name. The distinction between uppercase and lowercase is not significant in matching device names.

pbPassword
cbPassword   describe the device password that authorizes access to the specified device.

paramCode   describes the desired characteristics to which the floppy disk controller is to be initialized.

Code	Density	Sector Size	Compatibility
0	single	128	IBM Diskette 1
1	single	256	IBM Diskette 2
2	single	512	---
3	double	256	IBM Diskette 2D
4	double	512	B 22
5	double	1024	IBM Diskette 2D
6	reserved	---	---
7	double	256	B 21 (single-sided 5 1/4-in floppy)
8	double	256	B 21 (double-sided 5 1/4-in floppy)
9	double	512	MS-DOS 5 1/4-inch single sided (B21-2/3 only)
10	double	512	MS-DOS 5 1/4-inch dual sided (B21-2/3 only)

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	16
12	paramCode	2	
14	reserved	4	
18	pbDevSpec	4	
22	cbDevSpec	2	
24	pbPassword	4	
28	cbPassword	2	

Write

Description

The Write operation transfers an integral number of 128-, 256-, 512-, or 1024-byte sectors from memory to disk. Write returns only when the requested transfer is complete. The WriteAsync and CheckWriteAsync procedures are used to overlap computation and input/output transfer.

To accommodate programming languages in which Write is a reserved word, WriteFile is permitted as a synonym for the Write service.

Attempting to write beyond the end of the medium results in the return of status code 2 ("End of medium").

Procedural Interface

```
Write (fh, pBuffer, sBuffer, lfa,
       psDataRet): ErcType
```

where

fh	is a file handle returned from an OpenFile operation. The device must be open in modify mode.
pBuffer	is the memory address of the first byte of the buffer from which the data is to be written. The buffer must be word aligned.
sBuffer	is the count of bytes to be written from memory. It must be a multiple of the sector size (128, 256, 512, or 1024).
lfa	is the byte offset, from the beginning of the file, of the first byte to be written. It must be a multiple of the sector size (128, 256, 512, or 1024).
psDataRet	is the memory address of the word to which the count of bytes successfully written is to be returned.

Request Block

ssDataRet is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	36
12	fh	2	
14	lfa	4	
18	pBuffer	4	
22	sBuffer	2	
24	psDataRet	4	
28	ssDataRet	2	2

WriteAsync

Description

The WriteAsync procedure initiates the transfer
of an integral number of 128-, 256-, 512-, or
1024-byte sectors from memory to disk. The
CheckWriteAsync procedure must be called to check
the completion status of the transfer.

The information returned by Write with its
psDataRet argument and ErcType status is obtained
by CheckWriteAsync.

Procedural Interface

```
WriteAsync (fh, pBuffer, sBuffer, lfa, pRq,
           exchangeReply): ErcType
```

where

fh   is a file handle returned from an
                OpenFile operation. The file must
                be open in modify mode.

pBuffer   is the memory address of the first
                byte of the buffer from which the
                data is to be written. The buffer
                must be word aligned.

sBuffer   is the count of bytes to be written
                from memory. It must be a multiple
                of the sector size (128, 256, 512,
                or 1024).

lfa   is the byte offset, from the
                beginning of the file, of the first
                byte to be written. It must be a
                multiple of the sector size (128,
                256, 512, or 1024).

pRq   is the memory address of a 64-byte
                area to be used as workspace by
                WriteAsync.

exchangeReply
                is an exchange provided by the
                client process for the exclusive use
                of WriteAsync and CheckWriteAsync.

Request Block

The WriteAsync and CheckWriteAsync procedures are
procedural interfaces to the Write operation.
See the Write operation above.

SECTION 22
PRINTER SPOOLER MANAGEMENT

OVERVIEW

The printer spooler (simultaneous peripheral operation online) management facility provides direct and spooled printing to parallel (Centronics-compatible) and serial (RS-232C-compatible) printer interfaces.

Direct printing transfers text directly from application system memory to a parallel or serial printer interface of the local workstation. The local printer must be available before direct printing is activated.

In spooled printing, a queue entry is created for each printing request and entered in a queue managed by the queue manager. A printer spooler obtains a queue entry for printing when a printer is available. The user need not wait for a printer to be available to enter a printing request.

Direct and spooled printing are accessed by the printer spooler utilities described in the "Printer Spooler Utilities Overview" in the B 20 System Software Operation Guide, form 1148772. The reader should be familiar with that section before continuing in this section.

CONCEPTS

All printer spooler concepts are described in the "Printer Spooler Utilities Overview" section in the B 20 System Software Operation Guide, form 1148772. The following operations are contained in this section.

ConfigureSpooler, which sets or changes the printer spooler's configuration, and

SpoolerPassword, which sends a file passwords to the printer spooler.

Printer Spooler Configuration

When the printer spooler is installed, it reads a spooler configuration file designated by the user.

The spooler configuration file at printer spooler installation must contain at least the code of each printer channel to be controlled by the printer spooler. Additional information required for each printer can be supplied to the printer spooler in either of two ways:

in the spooler configuration file at printer spooler installation, or

dynamically through the ConfigureSpooler operation.

The additional information required for each printer is:

the name of the printer,

the name of the scheduling queue,

the printer configuration file specification,

the priority of the process that controls the printer, and

whether to print a banner page at the beginning of each file.

Sending a Password

If the security mode is specified in a printing request, the printer spooler pauses before printing the file and waits for receipt of a password. The password can be sent to the printer spooler in either of two ways:

> by the operator invoking the Spooler utility and typing the password at the local printer, or

> by a process using the SpoolerPassword operation.

OPERATIONS: SERVICES

Printer spooler management provides the operations listed below.

ConfigureSpooler
> sets or changes the spooler's configuration.

SpoolerPassword
> sends a file password to the printer spooler.

ConfigureSpooler

Description

> The ConfigureSpooler service sets or changes has printer spooler's configuration. A printer can be added or deleted from a printer spooler. To add a printer, the printing queue associated with the printer must be defined to the queue manager and the printer channel must be defined to the printer spooler during the printer spooler's installation.

Procedural Interface

>   ConfigureSpooler (channel, pbPrinterName,
>   cbPrinterName, pbQueueName,
>   cbQueueName, pbConfigureFile,
>   cbConfigureFile, priority,
>   fBanner): ErcType

> where

> channel   is a single-character code that specifies the printer channel to which the printer is connected:

>   0 is the parallel channel
>   A is channel A
>   B is channel B

>   .   .
>   .   .
>   .   .

> pbPrinterName
> cbPrinterName
>   describe the name of the printer to be added. A 0 means the printer connected to the channel is deleted.

> pbQueueName
> cbQueueName   describe the name of the .scheduling queue associated with the printer. The name must match a queue name defined for the system.

> pbConfigureFile
> cbConfigureFile
>   describe the file specification of the printer configuration file.

22-4

priority   is the priority (10-24, with 10 the highest) of the printer spooler's control process for the printer. A priority lower than 128 (the default priority of the user program) ensures that the printer spooler does not impact the user program.

fBanner   is a flag that indicates whether a banner page is to be printed at the beginning of each file.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	188
12	channel	1	
13	pbPrinterName	4	
17	cbPrinterName	2	
19	pbQueueName	4	
23	cbQueueName	2	
25	pbConfigureFile	4	
29	cbConfigureFile	2	
31	priority	1	
32	fBanner	1	
33	reserved	3	

SpoolerPassword

Description

> The SpoolerPassword service sends a file password
> to the printer spooler. If the printer spooler
> is in the security mode, it waits for the
> password before it proceeds to open and read the
> protected file.

Procedural Interface

> SpoolerPassword (pbPrinterName, cbPrinterName,
>   pbPassword, cbPassword): ErcType
>
>
> pbPrinterName
> cbPrinterName
>   describe the name of the printer.
>
>
> pbPassword
> cbPassword   describe the password.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	189
12	reserved	6	
18	pbPrinterName	4	
22	cbPrinterName	2	
24	pbPassword	4	
28	cbPassword	2	

SECTION 23
VIDEO MANAGEMENT

OVERVIEW

The video subsystem provides a highly flexible means for the display of alphanumeric and (limited) graphic information by the application system in the primary application partition. The video hardware uses DMA to continuously refresh the image on the screen, thus ensuring a flicker-free image. The video hardware reads characters and attributes from memory. It then converts them from the extended ASCII (8-bit) memory representation to a pattern of illuminated dots (pixels) that it displays on the screen. During this conversion, the video hardware references a translation table (font) that is part of the video hardware. In some models of workstation, the font can be modified by software.

Video Attributes

Video attributes control the visual presentation of characters on the screen. There are three kinds of video attributes: screen, line, and character.

o   Screen attributes control the presentation of the entire screen. Examples are: blank, reverse video (dark characters on a light background), half-bright, number of characters per line (80 or 132), and the presence or absence of character attributes.

o   Line attributes control the presentation of a single line. Examples are: cursor position, double-height characters, and double-width characters.

o   Character attributes control the presentation of a single character. Examples are: reverse video, blinking, half-bright, underlining, bold, superscript, and subscript.

Video Software

The video software provides several features (multiple frames, scrolling of each frame) that enhance the capabilities of the video subsystem. To the video software, the screen consists of a number of separate, rectangular areas called frames. A frame can have any desired height and width (up to those of the entire screen). Each frame can be scrolled up or down independent of other frames.

Hierarchy of Video Software

Three levels of video software control the video subsystem:

o   Video Display Management (VDM). The Video Display Management facility provides direct control over the video hardware.

o   Video Access Method (VAM). The Video Access Method provides direct access to the characters and character attributes of each frame. The Video Access Method includes explicit control of scrolling.

o   Sequential Access Method (SAM). The Sequential Access Method provides device-independent access to devices such as the printer, files, keyboard, as well as the video display. The Sequential Access Method provides automatic scrolling. Video-specific extensions to the Sequential Access Method provide direct cursor addressing, control of character attributes, etc.

CONCEPTS

Video Capabilities

The several models of workstation have varying video capabilities: basic, standard, or advanced.

Users who access the video subsystem at the Video and Sequential Access Method levels (but not the Video Display Manager level), and who only use the basic video capability, are assured compatibility among the different models.

Basic

Basic video capabilities are provided by the B21 workstation. These capabilities are characterized by: an 80-character by 28-line screen, one cursor on the screen, a 256 character set that cannot be modified by software, and a screen split horizontally into multiple frames.

Standard

Standard video capabilities are provided by the B22 family of workstations. These capabilities are characterized by a 34-line screen, a software-selectable 80- or 132-character line, one cursor per line, a 256 character set that can be dynamically modified by software, and a screen split horizontally and/or vertically into multiple frames that can overlap each other.

Standard Video Capability

Characters displayed on the screen are stored in a contiguous area called the character map. The physical memory address and size of the character map are loaded into the video DMA channel. The character map must start at a word boundary and must be completely contained in the first 128k bytes of memory. There is a default 6800-byte character map in the System Image. Alternatively, the application system in the primary application partition can relocate the character map to an area of long-lived memory it allocated.

The size of this character map depends on:

o   the number of characters per line (80 or 132),

o   the number of lines per screen (1 to 34), and

o   the presence or absence of character attributes.

Video Attributes

Video attributes control the visual presentation of characters on the screen. The three kinds of video attributes are screen, line, and character.

Screen Attributes. Screen attributes control the presentation of the entire screen. The screen attributes, specified in the Screen Attribute Register in the video hardware, are blank, reverse video (dark characters on a light background), half-bright, number of characters per line (80 or 132), and the presence or absence of character attributes. The number of lines (1 to 34) displayed on the screen is determined implicitly by the size of the character map loaded into the video DMA register.

Line Attributes. Line attributes control the presentation of a single line. They are specified in the character map in the word that precedes the first character of the line. The standard line attribute is cursor position.

Character Attributes. Character attributes control the presentation of a single character. Character attributes can be present or absent, depending on the value of a screen attribute. If character attributes are present, then each character has a 4-bit character attribute field. The 4-bit character attribute field specifies the presence or absence of four attributes: reverse video, blinking, half-bright, and underlining,

Video Refresh

The video hardware continuously refreshes the image on the screen, thus ensuring a flicker-free image. Video refresh is a hardware function that reads (using DMA) characters and line and character attributes from the character map in memory. It then converts them from the extended ASCII (8-bit) memory representation to a 10 by 15 bit array, and displays these bits on the screen as a pattern of illuminated dots (pixels).

Font RAM. During the conversion from a memory representation to a bit array, the standard video hardware references a translation table (font) located in the font RAM. The font RAM, part of the video hardware, contains a 10 by 15 bit array for each of the 256 characters in the character set. The font RAM can be modified by software.

Cursor RAM

A cursor RAM allows software to specify a 10 by 15 bit array and display these bits as a pattern of pixels in place of the standard cursor (a blinking underline). The cursor bit array is superimposed on the character and blinks.

Style RAM

> To allow access to the additional character attributes, the 4-bit character attribute field is interpreted as an index into the style RAM. The style RAM contains 16 entries, each of which specifies the presence or absence of each of the video attributes.

Basic Video Capability

> The B21 has basic video capability only, in contrast to the standard video capability of the B22.
>
> Characters displayed on the screen are stored in a contiguous area of memory called the character map. The physical memory address and size of the character map are loaded into the video DMA channel. The character map must be completely contained in the first 64k bytes of memory. There is a default 2784-byte character map in the System Image. Alternatively, the application system can relocate the character map to an area of long-lived memory that it allocated. The size of this character map depends on the number of lines per screen (1 to 28).

Video Attributes

> Video attributes control the visual presentation of characters on the screen. There are two kinds of video attributes: screen and character.
>
> **Screen Attributes.** Screen attributes control the presentation of the entire screen. Screen attributes, processed by the video software, are blank, reverse video (dark characters on a light background), half-bright, and cursor position. The number of lines (1 to 28) displayed on the screen is determined implicitly by the size of the character map loaded into the video DMA register.
>
> **Character Attributes.** Character attributes control the presentation of a single character. Character attributes are 8-bit bytes that are

embedded in the character map and specify the presence or absence of five attributes: reverse video, blinking, half-bright, underlining, and special character. Character attributes are identified by having their high-order bit set and are limited to 16 per line. The special character attribute is used to display character codes 80h to 0FFh.

Video Refresh

The video hardware continuously refreshes the image on the screen, thus ensuring a flicker-free image. <u>Video refresh</u> is a hardware function that reads (using DMA) characters and character attributes from the character map in memory. It then converts them to a 9 by 11 bit array, and displays these bits on the screen as a pattern of illuminated dots (<u>pixels</u>).

Font ROM. During the conversion from a memory representation to a bit array, the basic video hardware references a translation table (font) located in the font ROM. The <u>font ROM</u>, part of the video hardware, contains a 9 by 11 bit array for each of the 256 characters in the character set.

Video Software

The video software provides several features (multiple frames, scrolling of each frame) that enhance the capabilities of the video subsystem. To the video software, the screen consists of a number of separate, rectangular areas called <u>frames</u>. A frame can have any desired height and width (up to those of the entire screen). The number of frames supported is a parameter supplied at system build; the default is 8. Each frame can be scrolled up or down independent of other frames.

Hierarchy of Video Software

Three levels of video software control the video subsystem: the Video Display Manager, the Video Access Method, and the Sequential Access Method.

Video Display Manager

The Video Display Management (VDM) provides direct control over the video hardware. With it, an application system can:

o   determine the level of video capability present,

o   load a new character font into the font RAM,

o   change screen attributes, such as reverse video and half-bright, while the screen is being video-refreshed,

c   stop video refresh (this is useful when moving or changing the size of the frames or the character map),

o   calculate the amount of memory needed for the character map based on the desired number of columns and lines, and the presence or absence of character attributes,

o   initialize each of the frames, and

o   initialize the character map.

Once the character map is initialized and video refresh started, the image on the screen is controlled by modifying the characters and attributes stored in the character map. This is best accomplished using the system common procedures of the Video Access Method and the object module procedures of the Sequential Access Method. If necessary, however, an application system can manipulate the image on the screen by writing directly into the character map. This is somewhat more efficient than using the procedures of the Video and Sequential Access Methods, but results in code that is not compatible among the several models of workstation.

Video Access Method

The Video Access Method (VAM) provides direct access to the characters and character attributes of each frame. The operations of VAM can:

o   put a string of characters anywhere in a frame,

23-8

o   specify character attributes for a string of characters,

o   scroll a frame up or down a specified number of lines,

o   position a cursor in a frame (each frame can have its own cursor except in workstations having only basic video capability), and

o   blank a frame (that is, set all character positions to blank, reset all character attributes, and eliminate any visible cursor from the frame).

The Video Access Method consists of a set of system common procedures.

Sequential Access Method

The Sequential Access Method (SAM) provides device-independent access to devices such as the printer, files, keyboard, as well as the video display. The video byte stream extensions to the Sequential Access Method support multiple frames, character attributes, and explicit positioning of characters in a frame, but do not support line attributes (other than cursor position). The Sequential Access Method recognizes a few special cursor-positioning characters including RETURN, NEXT PAGE, BACKSPACE, and TAB. When a special character or full line would cause the cursor to move below the bottom line of the frame, the Sequential Access Method automatically scrolls the frame and repositions the cursor.

Application System/Video Subsystem Interaction

To eliminate the need for user programming to support video display initialization, the B20 Executive performs initialization before invoking an application system. (See the B20 System Executive Reference Manual, form 1144474.) It also allows the workstation operator to use the Screen Setup command to respecify these video characteristics:

o   reverse video,

o   number of characters per line (80 or 132),

o   number of lines (1 to 34), and

o   the   presence   or   absence   of   character
     attributes.

When an application system is invoked, it
inherits the character font, the character map
(in system memory), and two frames (Command Frame
and Status Frame) from the Executive.   Video
refresh continues and the image on the screen
remains unchanged.

The application system can now update the image
on the screen by using the Video or Sequential
Access Methods or by directly manipulating the
content of the area of memory containing the
character map.

The   application   system   needs   to   use   the
operations of the Video Display Management
facility only if the character font, screen size,
frames, or provision for character attributes
must be changed during the execution of the
application system.

Video Control Block

The <u>Video Control Block</u> (VCB) contains all
information known to the OS about the video
display, including the location, height, and
width of each frame, and the coordinates at which
the next character is to be stored in the frame
by the Sequential Access Method.   The VCB is
located in system memory at an address recorded
in the System Common Address Table.   The VCB is
described in more detail below.

SYSTEM DATA STRUCTURES: VIDEO CONTROL BLOCK AND FRAME DESCRIPTOR

This section should be read after the "Video Display Management" and "Video Access Method" sections.

The Video Control Block (VCB) contains all information known about the video display, including the location, height, and width of each frame, and coordinates at which the next character is to be stored in the frame by the Sequential Access Method. The VCB is located in system memory at an address recorded at address 244h in the System Common Address Table. The content of the VCB is shown in Table 23-1.

The Video Control Block contains an array of frame descriptors. A frame descriptor is a component of the VCB and contains all information known about one of the frames. The number of frame descriptors in the VCB is specified at system build. The content of a frame descriptor is shown in Table 23-2.

Table 23-1. Video Control Block

Offset	Field	Size (Bytes)
0	level	1
1	fCharAttrs	1
2	fReverseVideo	1
3	fHalfBright	1
4	pMap	4
8	sMap	2
10	cFrames	1
11	cColsMax	1
12	cLinesMax	1
13	sLine	1
14	ibToAttrs	1
15	ibToChars	1
16	bSpace	1
17	SAR	2
19	pRgfLineDirty	4
23	pRgbRuns	4
27	iFrameCursor	1
28	pRgbRunsVirgin	4
32	rgbRgFrame	*

* 20 bytes for each frame defined
at system build.

where

level   is the level of video capability:

             0 = standard;
             2 = basic.

             Other values will identify future
             capability levels. It is set by the
             QueryVidHdw   and   ResetVideo
             operations.

fCharAttrs   is TRUE if the character map
             includes character attributes and
             the use of character attributes is
             enabled in the Screen Attribute
             Register in the video hardware. It
             is FALSE otherwise. The fAttr
             parameter to the ResetVideo
             operation is placed here. It is set
             by the ResetVideo operation.

fReverseVideo

is TRUE if the screen reverse video is enabled in the Screen Attribute Register in the video hardware. This causes the hardware to display dark characters on a light background. It is FALSE otherwise. It is initialized by the ResetVideo operation to FALSE and can be changed by the SetScreenVid-Attr operation.

fHalfBright is TRUE if screen half-bright is enabled in the Screen Attribute Register in the video hardware. This causes the hardware to display at half-bright. It is FALSE otherwise. It is initialized by the ResetVideo operation to FALSE and can be changed by the SetScreenVid-Attr operation.

pMap
sMap   are the memory address and size of the character map, which are provided as parameters to the InitCharMap operation.

cFrames   is the number of frames. This number is established at system build. The default is 8.

cColsMax
cLinesMax   are the height and width of the screen. These values are used by the InitVidFrame operation to verify frame coordinates and dimensions. They are set by the ResetVideo operation.

sLine   is the total number of bytes required to contain all the information for one line of the character map. This information includes line attributes, filler bytes needed by the video hardware, text characters, and character attributes (if specified). This field can be multiplied by the number of a line to compute the offset from pMap of the first byte of the line. It is set by the ResetVideo operation.

NOTE: The information given for the next two fields, ibToAttrs and ibToChars, is not meaningful for workstations with only basic video capability. Dependence on this information will result in software that is not compatible among workstation models.

ibToAttrs   is the number of bytes from the start of a line in the character map to the first byte of character attributes (this is only valid if fCharAttrs = TRUE). To compute the offset (from the beginning of the character map) of the character attribute field for the character at column iCol and line iLine:

1. multiply iLine by the sLine field of the VCB,

2. add the ibToAttrs field of the VCB, and

3. add the integer quotient of iCol divided by 2.

The character attributes for even column numbers are in bits 0-3 (low-order nibble) and for odd column numbers in bits 4-7 (high-order nibble). It is set by the Reset-Video operation.

ibToChars   is the number of bytes from the start of a line in the character map to the character 0 of the line. To compute the offset (from the beginning of the character map) of the character displayed at screen coordinates iLine and iCol:

1. multiply iLine by the sLine field of the VCB,

2. add the ibToChars field of the VCB, and

3. add iCol.

It is set by the ResetVideo operation.

23-14

bSpace   is the 8-bit character code that
                displays an empty character position
                on the screen.   This code is 0
                (null) in the standard Burroughs
                font.   The bSpace field of the VCB
                is set from a parameter to the
                ResetVideo operation and is used by
                the InitCharMap, ResetFrame, and
                ScrollFrame operations.

SAR   is an exact copy of the 16 bits that
                were last loaded into the Screen
                Attribute Register.

pRgfLineDirty
                is the memory address of an array of
                28 flags (bytes), one flag for each
                line. Each flag indicates whether
                or not video attributes are
                intermixed with the characters on
                that line (B21 only).

pRgbRuns   is the memory address of an array of
                16 x 28 words. The low-order byte
                of each word describes an attribute,
                and the high-order byte specifies
                the number of characters to which
                the attribute applies.   Sixteen
                words are used to describe each line
                on the video display (B21 only).

iFrameCursor
                stores the number of the frame in
                which the cursor is located. This
                field is updated whenever
                PosFrameCursor is called (B21 only).

pRgbRunsVirgin
                is the memory address of an array of
                16 words. The contents of this
                array represent a line that does not
                have attributes intermixed with the
                characters (B21 only).

rgbRgFrame   is the array of frame descriptors.
                It is set by the InitVidFrame
                operation and cleared by the
                ResetVideo operation.

Table 23-2. Frame Descriptor

Offset	Field	Size (bytes)
0	iLineStart	1
1	iColStart	1
2	cLines	1
3	cCols	1
4	iLineLeftOff	1
5	iColLeftOff	1
6	bBorderDesc	1
7	bBorderChar	1
8	bBorderAttr	1
9	iLinePause	1
10	iLineCursor	1
11	iColCursor	1
12	fDblWide	1
13	fDblHigh	1
14	reserved	6

where

iLineStart
iColStart   are the vertical and horizontal
                   screen coordinates of the upper left
                   corner of the frame. This is where
                   a character would go if the
                   PutFrameChars operation were called
                   with iLine and iCol = 0.

cLines
cCols   is the height and width of the
                  frame.

iLineLeftOff

iColLeftOff   is used by the Sequential Access Method to record the coordinates at which the next character is to be stored in the frame. The presence of this information in the VCB allows a successor application system to append to information displayed by its predecessor.

bBorderDesc   is a byte with bits 0-3 specifying a border just outside the frame on the corresponding side.

Bit	Side
0	Top
1	Right
2	Bottom
3	Left

The border is drawn when the InitCharMap operation is executed. The same character with the same character attributes (see the bBorderChar and bBorderAttr parameters of the InitVidFrame operation) is used for all sides and corners.

bBorderChar   is the character to use for borders when the InitCharMap operation is executed.

bBorderAttr   is the character attribute to use for borders when the InitCharMap operation is executed.

iLinePause   is used by the Sequential Access Method to determine when to prompt the workstation operator to press

the NEXT PAGE key after a new page
of text is scrolled onto the
screen. iLinePause indicates which
line (0-33) is "marked." iLinePause
is decremented whenever the marked
line is scrolled upward. When it is
decremented to 0, a message
prompting the user is displayed.
(See the complete description of
this in the subsection on
"Automatically Pausing Between Full
Frames of Text" in the "Sequential
Access Method" section.) If
iLinePause is set to 255 (0FFh), as
it is by the ResetFrame operation,
the functions described above are
suppressed.

iLineCursor
iColCursor   are the vertical and horizontal
coordinates within the frame at
which the visible cursor is
displayed. If iLineCursor and
iColCursor are each set to 255
(0FFh), there is no visible cursor
in the frame.

SECTION 24

VIDEO DISPLAY MANAGEMENT

OVERVIEW

The Video Display Management (VDM) facility provides direct control over the video hardware. With it, the application system in the primary application partition can:

determine the level of video capability present (basic, standard, and advanced video capabilities are described in the "Video Management" section),

load a new character font into the font RAM,

stop video refresh (useful when moving or changing the size of the frames or the character map),

change screen attributes, such as reverse video and half-bright, while the screen is being video-refreshed,

calculate the amount of memory needed for the character map based on the desired height and width of the characters, and the presence or absence of character attributes,

initialize each of the frames, and

initialize the character map.

Once the character map is set up and video refresh is started, the image on the screen is controlled by modifying the characters and attributes stored in the character map. This manipulation is best accomplished using the system common procedures of the Video Access Method and the object module procedures of the Sequential Access Method. If necessary however, the application system in the primary application partition can manipulate the image on the screen by writing directly into the character map. Writing directly is somewhat more efficient than using the procedures of the Video and Sequential Access Methods, but results in code that is not compatible among the several models of workstation.

CONCEPTS

Reinitializing the Video Subsystem

The varied capabilities of the video subsystem are initialized by a sequence of software operations. The application system in the primary application partition needs to reinitialize the video subsystem only if the desired state is not a capability of the Screen Setup command (described in the B20 System Executive Reference Manual, form 1144474.) An application system that reinitializes the video subsystem must include a sequence of software operations similar to the following. The application must:

1. Use the QueryVidHdw operation to determine the level of video capability present on the workstation in use.

2. Use the LoadFontRam operation to read the character font from a file to memory and then load this font into the font RAM, except in workstations having only the basic video capability. In workstations having advanced video #1 capability, the application system must load the cursor RAM and the style RAM using the LoadCursorRam and LoadStyleRam operations.

3. Use the ResetVideo operation to place the following information in the Video Control Block (described in the "Video Management" section):

   number of characters per line (80 or 132),

   number of lines per screen (1 to 34), and

   the presence or absence of character attributes.

4. Allocate a long-lived memory segment to use as the character map, if the use of the character map in system memory is unsatisfactory. When calling the AllocMemoryLL operation, the application system should specify the size computed by the ResetVideo operation. (See the "Memory Management" section.)

5.   Use the InitVidFrame operation to specify the screen coordinates and dimensions of one of the frames.

6.   Use the SetScreenVidAttr operation to set reverse video or half-bright, if desired.

7.   Use the InitCharMap operation to initialize the character map.

8.   Use the SetScreenVidAttr operation to initiate video refresh.

The application system can now display information by using the Video or Sequential Access Methods or by writing characters and attributes directly into the character map.

OPERATIONS: SERVICES

The Video Display Manager provides the operations
listed below.

InitCharMap   initializes the character map.

InitVidFrame   defines the screen coordinates
                       and dimensions of one of the
                       frames.

LoadFontRam   reads the character font from
                       the specified open file to the
                       specified memory area and then
                       transfers the font to the font
                       RAM.

QueryVidHdw   places information describing
                       the level of video capability
                       of the workstation in the
                       specified memory area.

ResetVideo   suspends video refresh, resets
                       all screen attributes, and
                       changes the values stored in
                       the Video Control Block to
                       reflect the specified
                       parameters.

SetScreenVidAttr   sets/resets a specified screen
                       attribute.

InitCharMap

Description

The InitCharMap service initializes the character map. The ResetVideo and InitVidFrame operations must be called first.

InitCharMap sets all character positions of the character map to blanks and resets all line and character attributes. It then places the border character at the character positions that define the border of the frames for which borders were requested. The border descriptor, border character, and border attributes of each frame are specified by the InitVidFrame operation and are stored in a frame descriptor of the Video Control Block. (The Video Control Block and frame descriptor are described in the "Video Management" section.)

Procedural Interface

InitCharMap (pMap, sMap): ErcType

where

pMap   is either 0 to indicate the use of the character map in system memory or is the memory address of a character map in long-lived memory.

sMap   is the size of the character map.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	76
12	reserved	6	
18	pMap	4	
22	sMap	2	

InitVidFrame

Description

The InitVidFrame service defines the screen coordinates and dimensions of one of the frames. InitVidFrame must be called at least once after the ResetVideo operation and before the InitCharMap operation is called. It can also be called while the video subsystem is in use to change a frame or to add a frame. The Video Control Block is updated to reflect the changed or added frame. (The Video Control Block and frame descriptor are described in the "Video Management" section.)

The screen coordinates of the upper left corner of the frame are specified by iColStart and iLineStart. The width and height of the frame are given by nCols and nLines, respectively. Frames can overlap, but they cannot exceed the screen dimensions.

Procedural Interface

```
InitVidFrame (iFrame, iColStart, iLineStart,
              nCols, nLines, borderDesc,
              bBorderChar, bBorderAttr,
              fDblHigh, fDblWide): ErcType
```

where

iFrame   is an integer that ranges from 0 to the number of frame descriptors in the Video Control Block minus 1. This identifies the frame to be acted upon and selects one of the frame descriptors of the Video Control Block for modification.

iColStart   is the column of the screen that corresponds to the leftmost column of the frame.

iLineStart   is the line of the screen that corresponds to the top line of the frame.

nCols   is the width of the frame in columns.

nLines   is the height of the frame in lines.

borderDesc   is a byte with bits 0-3 specifying a
              border just outside the frame on the
              corresponding side. Note that the
              border characters are in addition to
              the area defined by nCols and
              nLines.

              Bit   Side

               0   Top
               1   Right
               2   Bottom
               3   Left

              The border is drawn when the
              InitCharMap operation is executed.
              The same character and attributes
              (bBorderChar and bBorderAttr) are
              used for all sides and corners.
              Left and right borders are not
              permitted in workstations with only
              basic video capability.

bBorderChar   specifies the character code to use
              for the frame borders when drawn by
              the InitCharMap operation.

bBorderAttr   specifies the 4-bit character
              attribute field with which
              bBorderChar is to be displayed.

              To create complex borders, including
              corner characters, initialize a
              frame that defines the entire
              screen; then put the appropriate
              border characters and attributes
              into the character map (using the
              PutFrameChars and PutFrameAttrs
              operations; see the "Video Access
              Method" section).

fDblHigh   Must be false.

fDblWide   Must be false.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	10
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	75
12	iFrame	1	
13	iColStart	1	
14	iLineStart	1	
15	nCols	1	
16	nLines	1	
17	borderDesc	1	
18	bBorderChar	1	
19	bBorderAttr	1	
20	fDblHigh	1	
21	fDblWide	1	

LoadFontRam

Description

The LoadFontRam service reads the character font from the specified open file to the specified memory area and then transfers the font to the font RAM. The file must contain a 16-word entry for each of 256 characters. Thus the file is exactly 4096 words (8192 bytes) long. Word 0 of each 16-word entry must be 0; words 1 to 15 represent the 15 rows of the character from top to bottom. Only bits 9 to 0 (where bit 0 is the least significant) of each word are used and represent the pixels from left to right.

LoadFontRam only has effect in a video subsystem with standard or advanced video capabilities.

Procedural Interface

LoadFontRam (fh, pBuffer, sBuffer): ErcType

where

fh   is the file handle of an open file containing the character font.

pBuffer   is the memory address of the buffer to use in loading the font RAM.

sBuffer   is 8704. pBuffer/sBuffer describe the memory area to be used by LoadFontRam. The memory area must be completely contained in the first 128K bytes of memory and its size must be 8704 bytes.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	22
12	fh	2	
14	reserved	4	
18	pBuffer	4	
22	sBuffer	2	8704

QueryVidHdw

Description

The QueryVidHdw service places information describing the level of video capability of the workstation in the specified memory area. When writing software that must work on several models of workstation, use QueryVidHdw to determine the level of video capability present before calling the ResetVideo operation. The format of the returned data is shown below.

Offset	Field	Size (bytes)	Description
0	level	1	Level of video capability: 0 = standard; 2 = basic.
1	nLinesMax	1	Maximum number of lines (for example, 34).
2	nColsNarrow	1	For models of video hardware that permit a selection of line width (for example, 80/132 columns), nColsNarrow specifies the narrower (for example, 80) and nColsWide specifies the wider (for example, 132). For models with only one width, nColsNarrow is equal to nColsWide.

Offset	Field	Size (bytes)	Description
3	nColsWide	1	Wider line width (for example, 132).
4	bitMapLevel	1	Level of bit map capability: 0 = none; 1 = for B22 work-stations.
5	nPixelsHigh	2	Number of pixels high for this version of bit map.
7	nPixelsWide	2	Number of pixels wide for this version of bit map.
9	saGraphicsBoard	2	Only applies if bit map level is 1. Segment address of 64k memory segment assigned to Graphics Multibus Board.
11	ioPort	2	Only applies if bit map level is 1. This is the switch-selectable input/output port used to select a 64k segment within the Graphics Multibus Board memory.
13	reserved	87	

Procedural Interface

> QueryVidHdw (pBuffer, sBuffer): ErcType
>
> where
>
> | | |
> |---|---|
> | pBuffer | is the memory address of the buffer to which the video capability information is to be copied. |
> | sBuffer | is the size of the buffer. If sBuffer is too small, the data is truncated. |

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	21
12	reserved	6	
18	pBuffer	4	
22	sBuffer	2	

ResetVideo

Description

The ResetVideo service suspends video refresh, resets all screen attributes, and changes the values stored in the Video Control Block to reflect the specified parameters. Subsequent calls to the InitVidFrame operation are validated against the values in the Video Control Block. (The Video Control Block is described in the "Video Management" section.)

Any number of columns or lines can be specified if the video hardware permits specification of an equal or greater number of columns or lines. For example, a screen of 105 columns can be specified on a video subsystem that has an 80-column mode and a 132-column mode. In this case, the mode would be set to 132-column mode, the leftmost 105 columns would be used, and the rightmost 27 columns would be blank.

Three values (sLine, ibToAttrs, and ibToChars) are calculated and stored in the Video Control Block for the Video Access Method or equivalent user code. The rest of the Video Control Block is reset, notably the definitions of all frames. Also, the fExecScreen flag in the Application System Control Block is set to FALSE.

Procedural Interface

ResetVideo (nCols, nLines, fAttr, bSpace,
            psMapRet): ErcType

where

nCols   specifies the number of characters
            per line (1 to 132).

nLines   specifies the number of lines per
            screen (1 to 34).

fAttr   specifies whether the character map
            is to include character
            attributes. It is TRUE if character
            attributes are to be used; it is
            FALSE otherwise.

bSpace   specifies a character code that is
            blank in the font. This is used
            when the character map is

24-14

initialized by the InitCharMap operation, and by the ResetFrame and ScrollFrame operations. (See the "Video Access Method" section).

psMapRet   is the memory address of the word to which the required size of the character map is returned.

Request Block

ssMapRet is always 2.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	74
12	nCols	1	
13	nLines	1	
14	fAttr	1	
15	bSpace	1	
16	reserved	2	
18	psMapRet	4	
22	ssMapRet	2	2

SetScreenVidAttr

Description

The SetScreenVidAttr service sets and resets a specified screen attribute.

Procedural Interface

SetScreenVidAttr (iAttr, fOn): ErcType

where

iAttr   identifies the screen attribute.

Value	Screen Attribute
0	reverse video
1	video refresh
2	half-bright

fOn   is TRUE to turn the specified screen attribute on and FALSE to turn it off.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	77
12	iAttr	2	
14	fOn	2	

SECTION 25
VIDEO ACCESS METHOD

OVERVIEW

The <u>Video Access Method</u> (VAM) provides direct access to the characters and character attributes of each frame. Its convenient interface provides the application system with independence from the position of the frame on the screen. In addition, the application system can be independent of the model of workstation in use as long as only the basic video capability is used. (The basic, standard, and advanced video capabilities are described in the "Video Management" section.)

VAM is a set of system common procedures.

Forms-Oriented Interaction

VAM is ideal for forms-oriented interaction; that is, interaction in which a form is displayed in a frame and the workstation operator enters data into the fields (blanks) of the form. Direct cursor addressing and modification of individual characters and character attributes support this interaction.

For example, the PutFrameAttrs operation is used to highlight the field to be entered next by setting reverse video for the range of character positions that compose the field. After the field is entered, the PutFrameAttrs operation is used again to reset the reverse video attribute on the character positions of the field.

Advanced Text Processing

VAM is also ideal for advanced text processing because it provides scrolling up and down of entire or partial frames. It is easy, for example, to scroll up the top four lines of a frame and insert a new line of text between the old fourth and fifth lines. During scrolling, character attributes scroll along with the text they affect.

OPERATIONS: PROCEDURES

The Video Access Method provides the operations listed below.

PosFrameCursor   establishes a visible cursor within the specified frame at the specified coordinates.

PutFrameAttrs   establishes the same character attribute for a range of character positions within a specified frame.

PutFrameChars   overwrites the specified character positions in the specified frame with the specified text string.

QueryFrameChar   returns a single character located in the character map at the specified coordinates of the specified frame.

ResetFrame   restores the frame to its initial state, that is, all character positions are blanked and all character attributes are reset.

ScrollFrame   scrolls the specified portion of the specified frame up or down by the specified number of lines.

PosFrameCursor

Description

The PosFrameCursor procedure establishes a visible cursor within the specified frame at the specified coordinates.

In a workstation with only basic video capability, PosFrameCursor erases any previously displayed cursor, even one in another frame.

Procedural Interface

PosFrameCursor (iFrame, iCol, iLine): ErcType

where

iFrame   specifies the frame.

iCol
iLine   specify the horizontal and vertical coordinates within iFrame at which to establish a cursor. To remove the cursor from a frame, both iCol and iLine must be specified as 255 (0FFh).

Request Block

PosFrameCursor is a system common procedure.

PutFrameAttrs

Description

The PutFrameAttrs procedure establishes the same
character attribute for a range of character
positions within a specified frame. The
character attribute is applied first left to
right and then top to bottom in the same manner
as characters are moved into a frame.

Procedural Interface

PutFrameAttrs (iFrame, iCol, iLine, attr,
                 nPos): ErcType

where

iFrame   specifies the frame.

iCol
iLine   specify the horizontal and vertical
                      coordinates within iFrame at which
                      to begin altering character
                      attributes.

attr   the low-order 4 bits of attr specify
                      the character attributes. For
                      workstations with basic or standard
                      video capabilities, the
                      interpretation of the bits is as
                      shown below.

Bit	Value	Attribute
0	1	Half-bright. (Note that if screen half-bright is set, the interpretation of the character attribute half-bright is to negate half-bright (that is, to display the character at full brightness.)
1	2	Underlining.

Bit	Value	Attribute
2	4	Reverse video. (Note that if screen reverse video is set, the interpretation of the character attribute reverse video is to negate reverse video (that is, to display a light character on a dark background.)
3	8	Blinking.

nPos   specifies the number of character positions whose character attributes are to be changed.

Request Block

PutFrameAttrs is a system common procedure.

PutFrameChars

Description

The PutFrameChars procedure overwrites the specified character positions in the specified frame with the specified text string. PutFrame-Chars does not cause the character attributes associated with the character positions to change and never causes scrolling.

Procedural Interface

PutFrameChars (iFrame, iCol, iLine, pbText,
               cbText): ErcType

where

iFrame   specifies the frame.

iCol
iLine   specify the horizontal and vertical
                coordinates within iFrame at which
                the first character of the text
                string is to be moved.

pbText
cbText   describe the text string to be moved
                into the character map.

Request Block

PutFrameChars is a system common procedure.

QueryFrameChar

Description

The QueryFrameChar procedure returns a single character located in the character map at the specified coordinates of the specified frame.

Procedural Interface

QueryFrameChar (iFrame, iCol, iLine,
                pbRet): ErcType

where

iFrame   specifies the frame.

iCol
iLine   specify the horizontal and vertical coordinates within iFrame of the character to be returned.

pbRet   is the memory address of the byte to which the character is to be returned.

Request Block

QueryFrameChar is a system common procedure.

ResetFrame

Description

The ResetFrame procedure restores the frame to its initial state, that is, all character positions are blanked and all character attributes are reset. The visible cursor of the frame is disabled (the iLineCursor and iColCursor fields of the frame descriptor of the Video Control Block are set to OFFh). (The Video Control Block and frame descriptor are described in the "Video Management" section.) The coordinates at which the Sequential Access Method is to place the next character are set to the upper left corner of the frame (the frame descriptor fields iLineLeftOff and iColLeftOff are set to 0).

To support the Sequential Access Method's ability to request confirmation before scrolling information off the top of the frame, the first line of the frame is marked, unless pausing is disabled (the frame descriptor field iLinePause is set to 0, unless its previous value was OFFh).

Procedural Interface

ResetFrame (iFrame): ErcType

where

iFrame   specifies the frame.

Request Block

ResetFrame is a system common procedure.

ScrollFrame

Description

The ScrollFrame procedure scrolls the specified portion of the specified frame up or down by the specified number of lines. Vacated lines are replaced by blank lines. The portion to scroll begins at iLineStart and extends down to, but does not include, iLineMax. It is scrolled up/down by cLines and the bottommost/topmost cLines lines of the scrolled area are filled with nulls (character code 0). fUp specifies the direction of the scroll. A value of 0FFh for iLineStart or iLineMax specifies an imaginary line just below the bottom of the frame.

For example, to scroll an entire frame up by one line, specify:

```
iLineStart = 0
iLineMax = 0FFh
cLines = 1
fUp = TRUE
```

To open a two-line space at line 4 (that is, lines 4 and 5 become blank) by scrolling the frame down, specify:

```
iLineStart = 4
iLineMax = 0FFh
cLines = 2
fUp = FALSE
```

To close the two-line space again, by scrolling the frame up (leaving the bottom two lines blank), specify:

```
iLineStart = 4
iLineMax = 0FFh
cLines = 2
fUp = TRUE
```

Procedural Interface

> ScrollFrame (iFrame, iLineStart, iLineMax,
>   cLines, fUp): ErcType
>
> where
>
> iFrame   specifies the frame.
>
> iLineStart   is the line at the top of the area
>   to scroll.
>
> iLineMax   is the line just below the area to
>   scroll.
>
> cLines   is the number of lines by which to
>   scroll.
>
> fUp   specifies the direction of the
>   scroll. It is TRUE for scroll up or
>   FALSE for scroll down.

Request Block

> ScrollFrame is a system common procedure.

SECTION 26
KEYBOARD MANAGEMENT

OVERVIEW

The keyboard management facility enables the application system in the primary application partition to control the keyboard.

Physical Keyboard

The 98-key keyboard includes ten special function keys and eight keys with LEDs. The keyboard is <u>unencoded</u>, that is, pressing or releasing a key causes unambiguous information to be transmitted from the 8048 microprocessor in the keyboard to keyboard management.

Consider this sequence: press the SHIFT key (to the left of the Z), press the A, release the A, release the SHIFT. An encoded keyboard would transmit only one item of information, the code for uppercase A. The B20 unencoded keyboard, however, transmits four items of information, one for each key transition. It also differentiates the depression/release of the left SHIFT key from the depression/release of the right SHIFT key.

Although this Manual refers to the keys by the standard symbols engraved on them, the meaning of each key is completely under the control of the application system in the primary application partition.

Keyboard Modes: Unencoded and Character

The application system in the primary application partition can request input from the keyboard in either of two modes: unencoded or character.

In unencoded mode, the application system receives an indication of each key depression and release. This mode provides maximum flexibility. With unencoded mode, an application system can, for example, use any key as a SHIFT key, provide a hierarchy of SHIFT keys, and make decisions based on how long a key remains depressed. These are only three of many possibilities. The Editor makes extensive use of the flexibility afforded by unencoded mode. See the B20 Systems Editor Reference Manual, form 1148673 and especially note the description of the MOVE and COPY keys in the "Manipulating the Selection" section.

In character mode, the application system receives an 8-bit character code when a key other than SHIFT, CODE, LOCK, or ACTION is pressed. Character mode provides the application system with the same kind of information as a traditional n-key rollover encoded keyboard. However, even character mode provides greater flexibility than an encoded keyboard. As keyboard management converts the sequence of keyboard codes to 8-bit character codes, it accesses the Keyboard Encoding Table to direct its translation.

Keyboard Encoding Table

The Keyboard Encoding Table can be modified dynamically during application system execution, as well as at system build. This Table controls several aspects of the keyboard-code-to-character-code translation:

the character code to generate if the SHIFT key is/is not depressed,

whether the LOCK key has the effect of the SHIFT key for this key,

whether the key is Typematic (repeats),

the initial delay before beginning Typematic repeating, and

the frequency of Typematic repeating.

The standard Keyboard Encoding Table (see Appendix B) provides an 8-bit superset of the ASCII printable characters. All 256 8-bit character codes can be generated from the keyboard. Each of the first 128 character codes (and some of the second 128) can be generated either by pressing a single key or by depressing the SHIFT key while depressing another key. Depressing the CODE key while depressing another key causes the high-order bit to be set (80h to be inclusive ORed) in the character code that would otherwise be generated. Thus, the use of the CODE key (or the CODE and SHIFT keys) permits the generation of the remainder of the 256 character codes.

The ability to modify the Keyboard Encoding Table allows the keyboard to be customized without requiring the application system to support the complexity of directly interpreting the unencoded keyboard.

A typical requirement is to use the numeric pad keys as function rather than data entry keys. This requires that the application system distinguish, for example, between the 3 on the numeric pad and the 3 on the typewriter pad. Changing the entry for the 3 on the numeric pad in the Keyboard Encoding Table provides the selected unique code to the application system whenever that key is pressed. (Support of this function may also require changing the key cap engraving for the numeric pad 3 key.)

LED Keys

Seven of the eight keyboard LEDs are under application system control. The LED in the LOCK key is under control of the application system control in unencoded mode and control of keyboard management in character mode.

Submit Facility

The System Input Manager augments keyboard management by providing a submit facility. The submit facility permits a sequence of characters from a file to be substituted for characters typed at the keyboard. The use of submit files allows the convenient repetition of command sequences. For example, a submit file might be used to run the sequence of programs necessary to produce end-of-month reports.

One convenient way to use the submit facility is to use the Editor to prepare a submit file containing the same sequence of characters that would be typed to the desired programs. When this submit file is activated by a request from an application system or an Executive command, a character from the file is returned to the application system whenever it requests a character from the keyboard. (Since the System Input Manager always operates in character mode, this is not applicable to an application system that uses the keyboard in unencoded mode.)

Use of the submit facility does not preclude direct access to the keyboard. The application system can bypass an active submit file and read characters directly from the keyboard. This is necessary when the application system needs

confirmation that a physical action was performed. For example, if a submit file is used to produce a sequence of reports, the application system needs to accept confirmation from the keyboard, rather than from the submit file, that the correct report forms are loaded into the printer.

When requesting a character, an application system can specify that the character must come from the keyboard rather than the submit file. Also, a special sequence of characters (an <u>escape sequence</u>) in the submit file can cause input to be accepted temporarily directly from the keyboard. Pressing a special key causes the input source to revert to the submit file.

The System Input Manager has a complementary capability that records in a file all the characters typed at the keyboard, in addition to returning them to the application system requesting them. This file can be used as a record of all data typed at the workstation. Also, a file of this kind can be used as a submit file to repeat the same sequence of input characters to the same programs at a later time.

CONCEPTS

Physical Keyboard

The 98-key keyboard (see Figure 26-1) includes ten special function keys and eight keys with LEDs. The keyboard is unencoded, that is, pressing or releasing a key causes unambiguous information to be transmitted from the 8048 microprocessor in the keyboard to keyboard management.

When a key is depressed or released, the 8048 microprocessor in the keyboard transmits a sequence of bytes to indicate all keys currently depressed. The seven low-order bits of each byte identify the key. The high-order bit is 0 in all bytes except the last of the sequence; it is set in the last byte to indicate the end of the sequence. A special code is transmitted to indicate that the last key was released and that no keys remain depressed.

Keyboard management remembers which keys are depressed. When it receives a byte sequence from the keyboard microprocessor, it compares the keys now reported as depressed to the ones it remembers as depressed. The differences are the keys depressed/released. This information is represented in an 8-bit byte for each key depression/release. The seven low-order bits identify the key; the high-order bit is 0 to indicate key depression and 1 to indicate key release.

Keyboard Modes: Unencoded and Character

An application system in the primary application partition can use the SetKbdUnencodedMode operation to specify in which mode the ReadKbd and ReadKbdDirect operations are to function. There are two modes: unencoded and character.

In unencoded mode, the application system receives an indication of each key depression and release. In this mode, the 8-bit byte, the keyboard code, returned by the ReadKbd or ReadKbdDirect operation identifies the key in the seven low-order bits; the high-order bit is 0 to indicate key depression and 1 to indicate key release. Appendix C specifies the 7-bit code generated for each key of the physical keyboard.
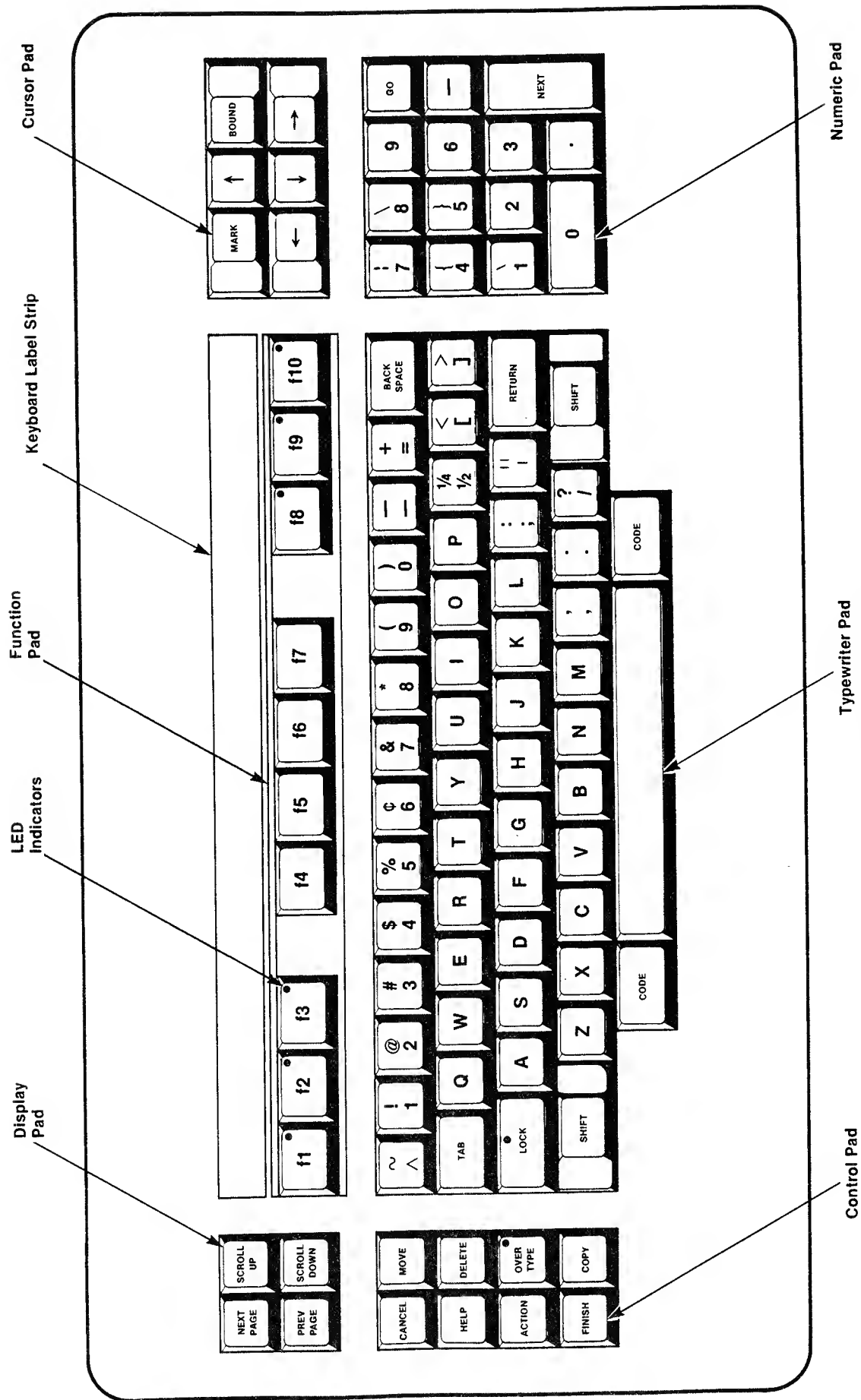
Figure 26-1. Keyboard

In character mode, the default mode, the 8-bit byte, the character code, returned by the ReadKbd or ReadKbdDirect operation signifies the depression of a key other than SHIFT, CODE, LOCK, or ACTION. Depression of SHIFT, CODE, and LOCK does not generate a character code, but influences the character codes generated for other keys depressed concurrently. ACTION has a special, system-wide meaning. (See the "ACTION Key" section below.)

Type Ahead

Keyboard management has a type-ahead buffer that stores character codes (or keyboard codes, if in unencoded mode) not yet read by an application system. If the workstation operator types too many characters in advance of processing, the excess are discarded. When the application system reads beyond those characters that were buffered successfully, it receives status code 610 ("Type-ahead buffer overflow"). The size of the type-ahead buffer is usually 128 characters, but can be changed at system build. The content of the type-ahead buffer is discarded by the SetKbdUnencodedMode operation if the mode is actually changed and by the Chain and ErrorExit operations if the status code is abnormal (nonzero) (see the "Task Management" section).

ACTION Key

The ACTION key is a special kind of SHIFT key. It is processed specially, even in unencoded mode. The interpretation of all other keys is modified while ACTION is depressed.

Key combinations that include the ACTION key are processed independently of calls by the application system to the ReadKbd or ReadKbdDirect operation and are not affected by character or keyboard codes stored in the type-ahead buffer.

The key combination ACTION-FINISH terminates the execution of the current application system and invokes the Executive. The DisableActionFinish operation disables this feature.

The key combinations ACTION-A and ACTION-B invoke the Debugger if the Debugger is included in the Operating System at system build.

Some of the key combinations that include the ACTION key are available for interpretation by an application system. Depressing CANCEL, HELP, 0-9, or f1-f10 while ACTION is depressed causes the keyboard code for the key depressed in conjunction with ACTION to be remembered. This code is an action code and can be obtained by calling the ReadActionCode operation. This allows the application system to test for special operator intervention without preventing type ahead.

For example, the BASIC interpreter uses ACTION-CANCEL to interrupt computation without interfering with type ahead.

If the workstation operator types a second key combination that includes the ACTION key before the first is read by the ReadActionCode operation, the second action code supersedes the first.

Independence of Keyboard and Video

Keyboard management never communicates with the video subsystem. The application system is free to interpret each character as it chooses and to echo characters to the video subsystem when and how it chooses. Keyboard management attaches no special significance to keys such as FINISH, HELP, RETURN, or DELETE. Only ACTION has special significance.

Keyboard Encoding Table

Keyboard management converts the sequence of keyboard codes to 8-bit character codes using the Keyboard Encoding Table. (See Appendix B for the standard character set stored in the Keyboard Encoding Table.) The address of the the Keyboard Encoding Table is stored at address 270h in the System Common Address Table. This allows the Keyboard Encoding Table to be modified dynamically during application system execution, as well as at system build. This Table controls several aspects of the keyboard-code-to-character-code translation:

the character code to generate if the SHIFT key is/is not depressed,

whether the LOCK key has the effect of the SHIFT key for this key,

whether the key is Typematic (repeats),

the initial delay before begining Typematic repeating, and

the frequency of Typematic repeating.

See the B20 System Programmers and Assembler Reference Manual (Part 1), form 1148699 for detailed information on modifying the Keyboard Encoding Table.

Standard Character Set

The standard character set (see Appendix B) provides an 8-bit superset of the ASCII printable characters. All 256 8-bit character codes can be generated from the keyboard. Each of the first 128 character codes (and some of the second 128) can be generated either by pressing a single key or by depressing the SHIFT key while depressing another key. Depressing the CODE key while depressing another key causes the high-order bit to be set (80h to be inclusive ORed) in the character code that would otherwise be generated. Thus, the use of the CODE key (or the CODE and SHIFT keys) permits the generation of the remainder of the 256 character codes.

Submit Facility

The submit facility permits a sequence of characters from a file to be substituted for characters typed at the keyboard. The use of submit files allows the convenient repetition of command sequences.

A submit file can be activated by the SetSysInMode operation from an application system or by an Executive command. The submit file remains active until all its characters are read, an end-of-file escape sequence is read from it, or the application system calls the SetSysInMode operation again.

While a submit file is active, a character is read from the file and returned to the application system when the ReadKbd operation is called. After all characters are read from the submit file, it is automatically closed and subsequent ReadKbd operations read characters directly from the keyboard. The application system is not informed of the transition of input source from submit file to keyboard. This permits the use of submit files to be transparent to the application system. However, the QueryKbdState operation is available to an application system that needs to know whether a submit file is active.

Two circumstances can temporarily disable a submit file: the SetKbdUnencodedMode operation and a read-direct escape sequence (see the "Read-Direct Escape Sequence" section below).

If the application system sets unencoded mode by calling the SetKbdUnencodedMode operation, then the ReadKbd operation reads keyboard codes from the keyboard, not from the submit file. Thus, the submit facility is not available to application systems that use the keyboard in unencoded mode. When the application system calls the SetKbdUnencodedMode operation with the argument FALSE to set character mode, the submit file is reactivated and characters are again read from the submit file.

The submit file is also temporarily disabled when a read-direct escape sequence is read from the submit file.

The ReadKbdDirect operation is available to read from the keyboard at all times, regardless of whether a submit file is active.

The SetSysInMode operation can also specify recording mode. When recording mode is active, all characters typed at the keyboard and read in character mode by the ReadKbd operation (but not by the ReadKbdDirect operation) are written to a recording file, in addition to being returned to the application system of the ReadKbd operation. A recording file can later be used as a submit file to repeat the same sequence of input characters. A recording file and a submit file cannot be active simultaneously.

Submit File Escape Sequences

Certain sequences of characters (escape sequences) invoke special functions when read from a submit file. A submit file escape sequence consists of two or three characters. The first is the character code 03h (¢), which indicates the presence of an escape sequence. The second character of the escape sequence is a code to identify the special function. The third character, if present, is an argument to the special function. The permitted codes are shown in Table 26-1 below.

Table 26-1. Permitted Codes in Escape Sequences

Character	Code	Function
¢	03h	Two-character escape sequence that represents the character code 03h (¢). Since 03h is used to introduce escape sequences, this escape sequence (that is, two consecutive ¢) is the only way to represent the ¢ in a submit file.
1	31h	Three-character read-direct escape sequence (see below).
2	32h	End-of-file escape sequence. When this two-character escape sequence is read during a ReadKbd operation, the submit file is closed. The current and subsequent ReadKbd operations read characters directly from the keyboard. (This escape sequence is meaningful only in submit files that were created as recording files rather than through the Editor.)

The <u>read-direct escape sequence</u> is a three-character submit file escape sequence that causes the ReadKbd operation to read characters directly from the keyboard until a specified key is depressed. The third byte of the escape sequence specifies the key that is to terminate input from the keyboard. When the specified key is depressed, its keyboard code is <u>not</u> returned to the application system. Rather, the current and all subsequent ReadKbd operations read characters from the submit file (unless another escape sequence redirects the input source).

For example, it is frequently useful to have the operator enter data into a single field of an Executive command form (see the B20 System Executive Reference Manual, form 1144474) during the playing of a submit file. To accomplish this, the submit file should contain:

...

data for the previous field

0Ah (RETURN/NEXT)

the three-character escape sequence 03h, 31h, 0Ah (¢, 1, RETURN/NEXT)

0Ah (RETURN/NEXT)

data for the next field

...

When the escape sequence is read from the submit file, the cursor is blinking in the leftmost character position of the field that is to be entered manually. The operator then enters the desired data into the field and presses either the RETURN or the NEXT key (symbolized by RETURN/NEXT). Pressing RETURN/NEXT resumes the execution of the submit file but control is not returned to the application system. The second RETURN/NEXT in the submit file ends the entry of data into the field and advances to the next field of the form.

As another example: it may be useful to have the operator enter data into all the fields of a form during the playing of a submit file. To accomplish this, include the four characters:

```
03h, 31h, 1Bh, 1Bh
```

in the submit file. This causes all characters
except GO (1Bh) to be read from the keyboard.
When the operator completes the form and presses
GO, the GO read from the keyboard resumes the
playing of the submit file. The GO in the submit
file (the 1Bh following the three-character
escape sequence) completes the processing of the
form.

Application System Termination

When an application system terminates (because of
the Chain, Exit, or ErrorExit operations, or the
ACTION-FINISH key combination):

> if the keyboard was in unencoded mode, it is
> reset to character mode and the content of
> the type-ahead buffer is discarded,

> the ACTION-FINISH feature is reenabled, and

> the action code, if any, is discarded.

In addition, if the application system terminates
abnormally (because of the Chain or ErrorExit
operations with a nonzero status code, or ACTION-
FINISH):

> the content of the type-ahead buffer is
> discarded, and

> the submit or recording file is closed.

Termination of the application system does not
affect the keyboard LEDs. However, the Executive
resets the LEDs when it is called.

OPERATIONS: SERVICES

Keyboard management provides the operations listed below.

Beep   actives an audio tone for one-half second.

CheckpointSysIn   writes the content of the current, partially filled, output buffer to the recording file if the System Input Manager is in recording mode.

DisableActionFinish
                    disables   OS interpretation of ACTION-FINISH.

QueryKbdLeds   returns the state (on/off) of the eight keyboard LEDs.

QueryKbdState   returns the status of the keyboard and the System Input Manager to a structure provided by the application system.

ReadActionCode   returns the action code, if any, and resets the indication that an action code is available.

ReadKbd   reads one character from the keyboard, or from a submit file if one is active.

ReadKbdDirect   reads one character code (or keyboard code, if in unencoded mode) from the keyboard.

SetKbdLed   turns on/off one of the keyboard LEDs.

SetKbdUnencodedMode
                    selects unencoded or character mode.

SetSysInMode   changes the state of the System Input Manager.

Beep

Description

The Beep service activates an audio tone for one-half second.

Procedural Interface

Beep: ErcType

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	52

CheckpointSysIn

Description

The CheckpointSysIn service writes the content of the current, partially filled, output buffer to the recording file if the System Input Manager is in recording mode. If the System Input Manager is in normal or submit mode, no action occurs.

Procedural Interface

CheckpointSysIn: ErcType

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	68

DisableActionFinish

Description

The DisableActionFinish service permits an application system in the primary application partition to disable OS interpretation of the ACTION-FINISH key combination.

Normally, the operator can terminate the current primary application system by simultaneously depressing the ACTION and FINISH keys. However, it is highly undesirable to terminate the execution of certain types of application systems. DisableActionFinish permits such application systems to disable OS interpretation of ACTION-FINISH.

Procedural Interface

DisableActionFinish (fDisable): ErcType

where

fDisable   disables ACTION-FINISH if TRUE or enables ACTION-FINISH if FALSE.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	67
12	fDisable	2	

QueryKbdLeds

Description

The QueryKbdLeds service returns the state (on/off) of the eight keyboard LEDs.

Procedural Interface

QueryKbdLeds: (pLEDsRet): ErcType

where

pLEDsRet   is the memory address of a byte to which the state is returned.

Bit	Key
0 (low)	f10
1	f9
2	f8
3	f3
4	f2
5	f1
6	LOCK
7	OVERTYPE

Request Block

sLEDsRet is always 1.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	55
12	reserved	6	
18	pLEDsRet	4	
22	sLEDsRet	2	1

QueryKbdState

Description

The QueryKbdState service returns the status of the keyboard and the System Input Manager to a structure provided by the application system in the primary application partition.

Procedural Interface

QueryKbdState (pKbdDescRet): ErcType

where

pKbdDescRet   is the memory address of a 16-byte keyboard descriptor area to which the status of the keyboard and the System Input Manager are returned.

Offset	Field	Size (bytes)
0	fUnencodedMode	1
1	sysInMode	1
2	fhSysIn	2
4	reserved	12

where

fUnencodedMode
  is character mode if FALSE or unencoded mode if TRUE.

sysInMode
  0 = normal mode (neither submit nor recording mode is active);
  1 = recording mode (a copy of keyboard input is being written to the file specified by fhSysIn);
  2 = submit mode (input is being read from the file specified by fhSysIn);

  3 = escaped submit mode (the submit file specified by fhSysIn is in use, but a read-direct escape sequence was read from that file causing input to be read directly from the keyboard until a special key is pressed).

fhSysIn
> is the file handle of the
> currently open submit or recording
> file. If sysInMode is 0 (normal
> mode), fhSysIn is not meaningful.

Request Block

sKbdDescRet is always 16.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	58
12	reserved	6	
18	pKbdDescRet	4	
22	sKbdDescRet	2	16

ReadActionCode

Description

The ReadActionCode service returns the action code, if any, and resets the indication that an action code is available. If no action code is available, ReadActionCode returns status code 609 ("No action code available").

Procedural Interface

ReadActionCode (pCodeRet): ErcRet

where

pCodeRet   is the memory address of a byte to which the keyboard code of the key that was depressed while the ACTION key was depressed is to be returned.

Request Block

sCodeRet is always 1.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	60
12	reserved	6	
18	pCodeRet	4	
22	sCodeRet	2	1

ReadKbd

Description

The ReadKbd service reads one character from the keyboard. If a submit file is currently active, ReadKbd reads the character from that file instead of from the keyboard.

Procedural Interface

ReadKbd (pCharRet): ErcType

where

pCharRet   is the memory address of a byte to which the character is to be returned.

Request Block

sCharRet is always 1.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	53
12	reserved	6	
18	pCharRet	4	
22	sCharRet	2	1

ReadKbdDirect

Description

The ReadKbdDirect service reads one character code (or keyboard code, if in unencoded mode) from the keyboard. ReadKbdDirect never reads from a submit file. Special modes permit testing for the presence of a character in the type-ahead buffer.

Procedural Interface

ReadKbdDirect (mode, pCharRet): ErcType

where

mode   is one of the following codes:

Code	Description
0	wait until a character code (or keyboard code, if in unencoded mode) is available, then return it.
1	if a character code (or keyboard code, if in unencoded mode) is currently available, return it. If no character code or keyboard code is available, return status code 602 ("No character available").
2	wait until a character code (or keyboard code, if in unencoded mode) is available, then return a copy of it but do not remove it from the type-ahead buffer. A subsequent ReadKbdDirect or ReadKbd operation reads the same character code or keyboard code again.

Code	Description
----	-----------

3	if a character code (or keyboard code, if in unencoded mode) is available, return a copy of it but do not remove it from the type-ahead buffer. If no character code or keyboard code is available, return status code 602 ("No character available").

pCharRet   is the memory address of a byte to which to return a character code or keyboard code.

Request Block

sCharRet is always 1.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	54
12	mode	2	
14	reserved	4	
18	pCharRet	4	
22	sCharRet	2	1

SetKbdLed

Description

The SetKbdLed service turns one of the keyboard LEDs on or off.

Procedural Interface

SetKbdLed (iLED, fOn): ErcType

where

iLED   is the identification of the LED to turn on/off.

iLED	Key
0	f10
1	f9
2	f8
3	f3
4	f2
5	f1
6	LOCK (only if the keyboard is in unencoded mode)
7	OVERTYPE

fOn   is on if TRUE or off if FALSE.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	56
12	iLED	2	
14	fOn	2	

SetKbdUnencodedMode

Description

The SetKbdUnencodedMode service selects unencoded or character mode. SetKbdUnencodedMode discards the content of the type-ahead buffer if the mode is actually changed.

Procedural Interface

SetKbdUnencodedMode (fOn): ErcType

where

fOn   is unencoded mode if TRUE or character mode if FALSE.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	57
12	fOn	2	

SetSysInMode

Description

The SetSysInMode service changes the state of the System Input Manager. SetSysInMode first closes the existing submit or recording file, if any, and then sets the specified mode using the specified file, if any.

Procedural Interface

SetSysInMode (iMode, fhSysIn): ErcType

where

iMode   is one of the following codes:

Code	Mode
0	normal mode (neither submit nor recording mode is active);
1	recording mode (a copy of keyboard input is to be written to the file specified by fhSysIn);
2	submit mode (input is to be read from the file specified by fhSysIn).

fhSysIn   is the file handle of the open file to use for the submit or recording file. The application system must make no further reference to this file. This is not used if iMode is 0 (normal mode).

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	59
12	fh	2	
14	iMode	2	

SECTION 27
COMMUNICATIONS MANAGEMENT

OVERVIEW

Each workstation, except the B21-1, includes an integral serial input/output (SIO) communications controller that supports two communications channels. Each channel can be used in asynchronous, character-synchronous, or bit-synchronous mode. Software support is provided at three levels.

Terminal Emulator,

Sequential Access Method, and

user-written Communication Interrupt Service Routine.

The Asynchronous Terminal Emulator (ATE), the X.25 Terminal Emulator, the 3270 Terminal Emulator, and the 2780/3780 RJE Terminal Emulator provide the ability to communicate with remote computers without requiring any user programming.

The Sequential Access Method supports full-duplex, asynchronous transmission, and X.25 transmission. See the "Sequential Access Method" section.

More specialized communications needs require a user-written Communication Interrupt Service Routine. See the "Interrupt Handlers" section and the B20 System Programmers and Assembler Reference Manual (Part 1), form 1148699.

OPERATIONS: PROCEDURES

Communications management provides the operations listed below.

LockIn   inputs from the SIO communications controller.

LockOut   outputs from the SIO communications controller.

LockIn

Description

> The LockIn procedure must be used in a B22 to input from the SIO communications controller. LockIn is necessary because the SIO communications controller supports two communications channels.
>
> If LockIn is not used, there may be unpredictable results on both the input/output operation being attempted, and the DMA operation in progress on the other Channel. In a cluster configuration, this almost certainly results in a system crash at the workstation in question, and possibly severe performance degradation throughout the cluster configuration.

Procedural Interface

> LockIn (bPort, bValueRet)
>
> where
>
> bPort   is the input/output port from which a byte value is to be read.
>
> bValueRet   is the byte value read.

Request Block

> LockIn is an object module procedure.

LockOut

Description

The LockOut procedure must be used in a B22 to
output from the SIO communications controller.
LockOut is necessary because the SIO
communications controller supports two
communications channels.

If LockOut is not used, there may be
unpredictable results on both the input/output
operation being attempted, and the DMA operation
in progress on the other Channel. In a cluster
configuration, this almost certainly results in a
system crash at the workstation in question, and
possibly severe performance degradation
throughout the cluster configuration.

Procedural Interface

LockOut (bPort, bValue)

where

bPort   is the input/output port to which a
            byte value is to be written.

bValue   is the byte value to be written.

Request Block

LockOut is an object module procedure.

SECTION 28
TIMER MANAGEMENT

OVERVIEW

Real-Time Clock

Each workstation has a Real-Time Clock (RTC). The RTC of the B22 family of workstations uses the power-line frequency (50 or 60 Hz) as a timing source. The RTC of the B21 family of workstations uses a crystal-controlled timing source.

Timer management uses the RTC to provide both the current date and time of day and the timing of intervals (in units of 100 ms). (For a cluster workstation without a local file system, the current date and time is maintained at the master workstation. For a cluster workstation with a local file system, the current date and time is maintained at both the master and cluster workstations.)

A client process can request that a message be sent to a specified exchange either once after a specified interval or repetitively with a specified constant interval between send operations. The first time a message is sent to an exchange can be up to 100 ms earlier than specified. Subsequent intervals are timed exactly.

Programmable Interval Timer

The workstations in the B22 family also have a second timer, a Programmable Interval Timer (PIT), that uses a 19.5 kHz crystal-controlled timing source to provide a resolution of 51.3 microseconds. The PIT is controlled by a 16-bit counter and therefore has a maximum interval of approximately three seconds.

Timer management uses the PIT to provide high-resolution, low-overhead activation of user pseudointerrupt handlers. A client process or interrupt handler can request that a pseudointerrupt handler be activated after a specified interval. Pseudointerrupt handlers are not available on the B21.

CONCEPTS

Simplified Date/Time Format

The simplified date/time format provides a compact representation of the date and the time of day. This precludes invalid dates and allows simple subtraction to compute the interval between two dates. The simplified date/time structure is shown in Table 28-1 below.

The date/time format is represented in 32 bits to an accuracy of one second. The high-order 15 bits of the high-order word contain the count of days since March 1, 1952. The use of a 15-bit field allows dates up to the year 2042 to be represented. The low-order bit of the high-order word is 0 to represent AM and 1 to represent PM. The low-order word contains the count of seconds since midnight/noon. Valid values are 0 to 43199.

The current date/time is maintained in the master workstation (for all the workstations of a cluster configuration) or in the standalone workstation. It can be accessed by the GetDateTime operation and modified by the SetDateTime operation.

Table 28-1. Simplified Date/Time Structure

Offset	Field	Size (Bytes)	Description
2	seconds	2	Count (0-43199) of seconds since last midnight/noon.
4	dayTimes2	2	Count (0-65535) of 12-hour periods since March 1, 1952 (0 = null date/time).

System Date/Time Format

If a client process executing on a master or standalone workstation needs to know the time to greater precision than one second, it can access the system date/time structure, the address of which is at address 240h in the System Common Address Table (described in Appendix E). The format of the system date/time structure is shown in Table 28-2 below.

Table 28-2. System Date/Time Structure

Offset	Field	Size (bytes)	Description
0	ticks	1	Counted down from 5 (50 Hz) or 6 (60 Hz) to 0.
1	hundredMsec	1	Count (0-9) of 100 ms since last second.
2	seconds	2	Count (0-43199) of seconds since last midnight/noon.
4	dayTimes2	2	Count (0-65535) of 12-hour periods since March 1, 1952 (0 = null date/time).

Expanded Date/Time Format

The ExpandDateTime and CompactDateTime operations convert between simplified date/time format and an expanded date/time format in which year, month, day of month, etc., are represented as discrete fields. The expanded date/time format is shown in Table 28-3 below.

Table 28-3. Expanded Date/Time Format

Offset	Field	Size (bytes)	Description
0	year	2	1952-2042 (0 = null date/time)
2	month	1	0 = Jan, 11 = Dec
3	day of month	1	1-31
4	day of week	1	0 = Sun, 6 = Sat
5	hour	1	0-23
6	minute	1	0-59
7	second	1	0-59

Timer Management Operations

There are five classes of timer management operations: date/time, format conversion, delay, Real-Time Clock, and Programmable Interval Timer.

Date/Time

The GetDateTime and SetDateTime operations access and modify the current OS date/time.

Format Conversion

The ExpandDateTime and CompactDateTime operations convert between simplified date/time format and an expanded date/time format in which year, month, day of month, etc., are represented as discrete fields. See Table 28-3 above.

Delay

The Delay operation allows a process to suspend execution for a specified interval (in units of 100 ms).

The OpenRTClock operation initiates the use of a client-process-provided data structure for control of complex <u>Real-Time Clock</u> (RTC) services. This data structure, the <u>Timer Request Block</u> (TRB), is shared by the client process and timer management. The CloseRTClock operation terminates the sharing of the TRB.

The TRB defines the <u>interval</u> after which a message is sent to a specified exchange. The message can be sent either once after the specified interval or repetitively with the specified constant interval between send operations. The message is the TRB itself.

The client process must acknowledge receipt of the TRB (as described below) before timer management will send the same TRB again. This ensures that system resources (link blocks) are not consumed by queueing the same TRB at the same exchange many times. The client process can also dynamically modify other fields of the TRB.

The format of a TRB is shown in Table 28-4 below.

Timer Management Operation. Every 100 ms, the timer management RTC interrupt handler performs the following sequence of operations on each active TRB. This sequence ensures that timer management will not send the same TRB again until the client process decrements the cEvents field to 0.

1. If the counter field is 0, do nothing.

2. Decrement the counter field by 1.

3. If the counter field has not become 0, do nothing more.

4. If the cEvents field is 0, send a message to the exchange specified by the exchResp field. The message is the TRB itself (not a copy of the TRB).

5. Increment the cEvents field by 1.

6. Copy the counterReload field to the counter field.

Table 28-4. Timer Request Block Format

Offset	Field	Size (bytes)	Description
0	counter	2	Decremented every 100 ms.
2	counterReload	2	Copied to counter field when counter reaches 0.
4	cEvents	2	Incremented when counter field reaches 0.
6	exchResp	2	Response exchange.
8	ercRet	2	Status code. Not used by timer management. Available for the client process.
10	rqCode	2	Request code. Not used by timer management. The client process should place a unique value in this field so that it can identify its TRB when it is received as a message.

"One-Shot" Timing. A client process should use the sequence below to initialize a TRB to time a single interval (a "one-shot" timer).

1. Set the counter field to 0.

2. Call the OpenRTClock operation.

3. Set the cEvents field to 0.

4. Set the counterReload field to 0.

5. Set the counter field to the desired interval.

Use the Wait or Check operation (specifying the exchange specified by the exchResp field) to receive the indication that the interval expired. (The Wait and Check operations are described in the "Interprocess Communication Management" section.) Remember that the RTC only operates in units of 100 ms. Thus, if the counter field is set to 3, the TRB can be sent to the exchResp exchange in as few as 200 ms or as many as 300 ms. To reuse the TRB to time another single interval, repeat the sequence above from step 3.

Repetitive Timing. A client process should use the sequence below to initialize a TRB for repetitive timing.

1. Set the counter field to 0.

2. Call the OpenRTClock operation.

3. Set the cEvents field to 0.

4. Set the counterReload field to the desired interval.

5. Set the counter field to the desired interval.

The first time that the TRB is sent to the exchResp exchange can be up to 100 ms earlier than specified. Subsequent intervals are timed exactly. This is guaranteed because the counter field of the TRB is decremented even if the client process has not finished processing the previous event. The cEvents field provides a continuous count of the events that have occurred but are not yet processed. If the client process is too slow, the count in the cEvents field becomes ever larger. Under these circumstances, the count in the cEvents field provides a measure of how far behind processing has fallen.

The client process should use the sequence below to process the TRB. This sequence avoids a race condition and yet processes the correct number of events.

1. Receive the indication that the interval expired by calling either the Wait or Check operation and specifying the exchange specified by the exchResp field.

2. If the cEvents field is 0, processing is complete; return to step 1.   (In this sequence, it is possible to receive a TRB in which cEvents is 0; thus it is necessary to perform this test before processing the event.)

3. Process the event.   Processing is application-specific.

4. Decrement the cEvents field by 1.   (It is not necessary to decrement the cEvents field in a single instruction unless the client process is keeping a count of events.)

5. Repeat the processing sequence from step 2.

Programmable Interval Timer

The Programmable Interval Timer (PIT), which is present in B22 workstations, is accessed through the SetTimerInt and ResetTimerInt operations.

The SetTimerInt operation establishes a pseudo-interrupt handler in the application system to receive a pseudointerrupt after a specified interval (in units of 51.3 microseconds).   The SetTimerInt operation specifies the memory address of a <u>Timer Pseudointerrupt Block</u> (TPIB) in user memory that must be allocated by the application system.

The format of a Timer Pseudointerrupt Block is shown in Table 28-5 below.

It is sometimes convenient to have a single pseudointerrupt handler service the pseudointerrupts associated with multiple TPIBs.   To do this, the pRqBlkRet field of each TPIB must point to the same 4-byte memory area and the SetTimerInt operation must be invoked for each TPIB.   The pseudointerrupt handler must examine this 4-byte memory area to determine which TPIB caused activation of the pseudo-interrupt handler.   Even when the pseudointerrupt

handler is serving only a single TPIB, pRqBlkRet must still point to an otherwise unused 4-byte memory area.

The ResetTimerInt operation terminates a previous SetTimerInt operation.

To understand the operation of a pseudointerrupt handler, read the "Interrupt Handlers" section.

Table 28-5. Timer Pseudointerrupt Block

Offset	Field	Size (bytes)	Description
0	linkField1	4	Used by the OS.
4	linkField2	4	Used by the OS.
8	pIntHandler	4	CS:IP of the entry point of the pseudointerrupt handler.
12	saData	2	Segment base address of the data segment to be used by the pseudo-interrupt handler.
14	cIntervals	2	Interval before the pseudointerrupt is to occur (in units of 51.3 micro-seconds).
16	pRqBlkRet	4	The memory address of 4 bytes into which the memory address of the TPIB is returned when the pseudointerrupt handler is invoked.
20	footPrint	2	Used by the OS.
22	delta	2	Used by the OS.
24	reserved	8	Used by the OS.

OPERATIONS: PRIMITIVES, PROCEDURES, AND SERVICES

Timer management operations are categorized by function in Table 28-6 below.

Table 28-6. Timer Management Operations by Function

<u>Date/Time</u>	<u>Delay</u>
GetDateTime	Delay
SetDateTime	
	<u>Real-Time Clock</u>
<u>Format Conversion</u>	
	CloseRTClock
CompactDateTime	OpenRTClock
ExpandDateTime	
	<u>Programmable Interval Timer</u>
	ResetTimerInt
	SetTimerInt

Date/Time

GetDateTime	returns the current data/time simplified date/time format.
SetDateTime	sets the date/time of the OS.

Format Conversion

CompactDateTime	converts from expanded date/time format to simplified date/time format.
ExpandDateTime	converts from simplified date/time format to expanded date/time format.

Delay

Delay	delays the execution of the client process for the specified interval.

Real-Time Clock

CloseRTClock	terminates the use of the specified TRB.

OpenRTClock	establishes a TRB between the client process and timer management.

Programmable Interval Timer

ResetTimerInt	terminates the TPIB initiated by a SetTimerInt call.
SetTimerInt	establishes a PIT pseudo-interrrupt handler.

CloseRTClock

Description

The CloseRTClock service terminates the use of the specified TRB.

The format of a TRB is shown in Table 28-4 above.

Procedural Interface

CloseRTClock (pRqTime): ErcType

where

pRqTime
sRqTime   describe a TRB that is currently open.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	50
12	reserved	6	
18	pRqTime	4	
22	sRqTime	2	

CompactDateTime

Description

The CompactDateTime procedure converts from
expanded date/time format to simplified date/time
format. Each field of the expanded date/time
format is verified. If the day of week does not
agree with the other fields, status code 2702
("Day and date disagree") is returned.

The expanded date/time format is shown in Table
28-3 above.

Procedural Interface

CompactDateTime (pExpDateTime,
               pDateTimeRet): ErcType

where

pExpDateTime

     is the memory address of an 8-byte
     expanded date/time block.

pDateTimeRet

     is the memory address of 4 bytes to
     which the simplified date/time
     format is to be returned.

Request Block

CompactDateTime is an object module procedure.

Delay

Description

The Delay procedure delays the execution of the client process for the specified interval.

Procedural Interface

Delay (n): ErcType

where

n   is the interval to delay (in units of 100 ms).

Request Block

Delay is a system common procedure.

ExpandDateTime

Description

The ExpandDateTime procedure converts from simplified date/time format to an expanded date/time format in which year, month, day of month, etc., are represented as discrete fields.

The expanded date/time format is shown in Table 28-3 above.

Procedural Interface

ExpandDateTime (dateTime,
                pExpDateTimeRet): ErcType

where

dateTime   is the 32-bit date/time in simplified format.

pExpDateTimeRet
                is the memory address of an 8-byte expanded date/time block to which expanded date/time format is to be returned.

Request Block

ExpandDateTime is an object module procedure.

GetDateTime

Description

The GetDateTime service returns the current date/time in the simplified date/time format.

Status code 46 ("Master workstation going down") is returned if a DisableCluster operation was performed. (See the "Cluster Management" section.) If status code 46 ("Master workstation going down") is returned, the low-order word of the structure pointed to by pDateTimeRet contains the time (in seconds) remaining before the master workstation goes down.

Procedural Interface

GetDateTime (pDateTimeRet): ErcType

where

pDateTimeRet

is the memory address of the 4-word structure to which the date/time is returned.

Request Block

sDateTimeRet is always 4.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	14
12	reserved	6	
18	pDateTimeRet	4	
22	sDateTimeRet	2	4

OpenRTClock

Description

The OpenRTClock service establishes a TRB between the client process that requests the timing services and timer management. The client process and timer management communicate by changing the fields in the TRB after an OpenRTClock call.

The format of a TRB is shown in Table 28-4 above.

Procedural Interface

OpenRTClock (pRqTime): ErcType

where

pRqTime   is the memory address of the client-process-provided TRB to be shared by the client process and timer management.

Request Block

sRqTime is always 12.

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	49
12	reserved	6	
18	pRqTime	4	
22	sRqTime	2	12

ResetTimerInt

Description

> The ResetTimerInt primitive terminates the TPIB initiated by a previous SetTimerInt call. ResetTimerInt is used only to cancel a previous SetTimerInt operation <u>before</u> the requested pseudointerrupt has occurred. The "No such TPIB" status code is returned if the pseudointerrupt has already occurred.
>
> The format of a TPIB is shown in Table 28-5 above.

Procedural Interface

> ResetTimerInt (pTPIB): ErcType
>
> where
>
> pTPIB   is the memory address of the TPIB to be terminated.

Request Block

> ResetTimerInt is a Kernel primitive.

SetDateTime

Description

The SetDateTime service sets the date/time of the OS.

Procedural Interface

SetDateTime (seconds, dayTimes2): ErcType

where

seconds   is the count (0-43199) of seconds since the last midnight/noon.

dayTimes2   is the count (0-65535) of 12-hour periods since March 1, 1952.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	51
12	seconds	2	
14	dayTimes2	2	

SetTimerInt

Description

The SetTimerInt primitive establishes a PIT pseudointerrupt handler and specifies the interval after which the PIT pseudointerrupt is to be generated. SetTimerInt can be called from a PIT pseudointerrupt handler to reestablish the PIT pseudointerrupt handler with a (possibly different) interval. Multiple pseudointerrupt handlers can use the PIT simultaneously.

The format of a TPIB is shown in Table 28-5 above.

Procedural Interface

SetTimerInt (pTPIB): ErcType

where

pTPIB   is the memory address of the TPIB.

Request Block

SetTimerInt is a Kernel primitive.

SECTION 29
INTERRUPT HANDLERS

OVERVIEW

An interrupt is an event that interrupts the sequential execution of processor instructions. When an interrupt occurs, the current hardware context (the state of the hardware registers) is saved. This context save is performed partly by the processor and partly by the Operating System. After the condition causing the interrupt is identified and acted upon, the context of the interrupted process (or another higher priority process) is restored and execution resumed as if the interrupt never occurred.

(The description that follows is not applicable to B21 workstations, since B21 workstations do not contain Multibus slots or support user-written interrupt handlers.)

External Interrupts

External (true) interrupts are caused by conditions that are external to the processor and are asynchronous to the execution of processor instructions.

In a standard workstation, eight of the interrupt levels are ordered in priority and controlled by the <u>8259A Programmable Interrupt Controller</u> (PIC). They can be masked (ignored) by the use of the processor interrupt-enable flag. They can also be selectively masked (that is, some recognized, some ignored) by programming the 8259A PIC.

Four of these eight interrupt levels are used for the standard device controllers of a workstation. The other four interrupt levels are available for the connection of device controllers that are installed in the Multibus-compatible card slots.

One interrupt level (with a higher priority than those controlled by the 8259A PIC) supports the critical error conditions:

o   write-protect violation,

o   nonexistent   memory   or   device   address
     reference,

o   memory parity error, and

o   power failure detection.

Internal Interrupts

Internal interrupts (traps) are caused by and are
synchronous with the execution of processor
instructions. The causes of internal interrupts
are:

o   an erroneous divide instruction,

o   the processor Trap Flag,

o   the INTO (interrupt on overflow) instruction,
     and

o   the INT (interrupt) instruction.

Device Handlers

Accommodation of user-written device handlers was
a major design goal of the Operating System. A
device handler can be part of either an
application system or a system service process.
Its interrupt handler can let the Kernel save the
process context (in which case it can be written
in FORTRAN or Pascal), or it can receive the
interrupt directly from the hardware.
Interprocess communication provides an efficient,
yet formal, interface from interrupt handler to
device handler and from device handler to
application system.

CONCEPTS

An interrupt is an event that interrupts the sequential execution of processor instructions. When an interrupt occurs, the current hardware context (the state of the hardware registers) is saved. This context save is performed partly by the processor and partly by the Operating System.

After the condition causing the interrupt is identified and acted upon, the OS Kernel either (1) restores the context of the interrupted process and resumes its execution, or (2) determines that a higher priority process is ready to execute, performs a context switch, and initiates execution of the higher priority process.

Interrupts can be nested, that is, a higher priority interrupt can interrupt the execution of an interrupt handler that is servicing a lower priority interrupt. When the higher priority interrupt handler completes its processing, execution of the lower priority interrupt handler resumes.

Interrupt Types

The processor has a simple yet versatile interrupt system. Each potential source of interrupt is assigned an interrupt type code. This is a number in the range 0-119 and is used to vector (direct) the interrupt to the appropriate interrupt handler. The Interrupt Vector Table begins at physical memory address 0 and contains a 4-byte entry for each interrupt type. Each 4-byte entry contains the logical memory address (CS:IP) of the first instruction to be executed when an interrupt of that type occurs.

The interrupt types are shown in Table 29-1.

Table 29-1. Interrupt Types

Interrupt Type Code	8259A PIC Level	Description	Interrupt Vector Address
0		Divide error trap.	00h
1		Single step trap (used by the Debugger).	04h
2		Nonmaskable external interrupts (write-protect violation, nonexistent memory or device address reference, memory parity error, or power failure detection.	08h
3		Breakpoint trap (used by the Debugger).	0Ch
4		Signed arithmetic overflow trap.	10h
5		Access to Kernel primitives.	14h
6		Access to system services.	18h
7		Access to system common procedures.	1Ch
8	0	Multibus devices.	20h
9	1	SIO communications controller.	24h
10	2	Multibus devices.	28h
11	3	Programmable Interval Timer.	2Ch

Table 29-1. Interrupt Types (Cont.)

Interrupt Type Code	8259A PIC Level	Description	Interrupt Vector Address
12	4	Printer, keyboard, Real-Time Clock, and high-speed mathematics coprocessor.	30h
13	5	Multibus devices.	34h
14	6	Multibus devices.	38h
15	7	Disk storage subsystem (floppy and Winchester).	3Ch
16-119		Available for software-generated interrupts and Multibus device interrupts using cascaded slave 8259A PICs on Multibus logic boards).	40h-1DCh

Interrupts

Interrupts are either external or internal.

External Interrupts

External interrupts are caused by conditions that are external to the processor and are asynchronous to the execution of processor instructions. There are two kinds of external interrupts: maskable and nonmaskable.

Maskable Interrupts. Maskable interrupts are given a priority and controlled by the 8259A Programmable Interrupt Controller (PIC). They can be masked (ignored) by the use of the processor interrupt-enable flag. They can also

be selectively masked (that is, some recognized, some ignored) by programming the 8259A PIC.

8259A Programmable Interrupt Controller. A "master" 8259A PIC is standard on each workstation and controls eight priority interrupt levels. Each interrupt level can be connected (wire ORed) to one or more device controllers or to a "slave" 8259A PIC. The use of slave 8259A PICs multiplies the number of external interrupt sources that can be uniquely identified and ordered in priority.

Four of the eight interrupt levels are used for the standard device controllers of a workstation.

The other four interrupt levels are available for the connection of device controllers that are installed in the Multibus-compatible card slots. These can be connected directly to the four available interrupt levels or can be multiplexed through the use of slave 8259A PICs installed on logic boards installed in the Multibus-compatible card slots.

Master workstations support large populations of cluster workstations by using one or more CommIOPs in the Multibus area. The use of other Multibus boards requires the implementation of user-written device handlers or interrupt handlers.

The 8259A PIC is a flexible hardware entity that can operate in a number of modes. The modes established by OS initialization are:

level (not edge) triggered,

fully nested (but not special fully nested),

fixed (not rotating) priority, and

not special mask.

CAUTION: Do not change the mode of the master 8259A PIC. Changing the mode causes the Operating System to malfunction in an unpredictable manner.

A device handler or interrupt handler can only perform the following operations on the master 8259A PIC (all other operations are forbidden):

o   They can read the Interrupt Mask Register (IMR) of the 8259A PIC; set or clear only the bit affecting the interrupt level serviced by the handler; and write the updated mask into the IMR. Processor interrupts must be disabled during this sequence.

o   They can read the Interrupt Request Register (IRR) or the Interrupt Service Register (ISR) of the 8259A PIC. Because reading either the IRR or the ISR requires issuing a command to the OCW3 register of the 8259A PIC to select the register to be read, processor interrupts must be disabled between selecting the register and reading it.

o   Raw interrupt handlers (but not mediated interrupt handlers) must issue either a specific or nonspecific End-Of-Interrupt (EOI) command to the 8259A PIC before returning from the raw interrupt handler to the point of interrupt.

CAUTION: Any other user programming of the master 8259A PIC causes the Operating System to malfunction in an unpredictable manner.

Slave 8259A PICs must be completely programmed by user code.

Nonmaskable Interrupts.   Nonmaskable interrupts (NMI) have a higher priority than maskable interrupts. NMIs cannot be masked through the use of the processor interrupt-enable flag; however, bits in the Input/Output Control Register allow each of the four conditions that cause NMIs to be masked individually. These conditions are:

o   write-protect violation,

o   nonexistent memory or device address reference,

o   memory parity error, and

o   power failure detection.

Internal interrupts (traps) are caused by and are synchronous with the execution of processor instructions. The causes of internal interrupts are:

o   an erroneous divide instruction (interrupt type 0),

o   the processor Trap Flag (interrupt type 1; single step),

o   the INTO (interrupt on overflow) instruction, if the processor Overflow Flag is set (interrupt type 4), and

o   the INT (interrupt) instruction (any interrupt type).

Pseudointerrupts

Pseudointerrupts are implemented in software rather than in hardware. In this sense, they are not really interrupts at all. However, they are similar to interrupts in that they cause an interrupt handler to be executed. An interrupt handler activated by a pseudointerrupt executes in the same environment and has the same responsibilities and privileges as an interrupt handler activated by a real interrupt.

As an example of the use of pseudointerrupts, the SetTimerInt operation (see the "Timer Management" section) establishes a Programmable Interval Timer pseudointerrupt handler to service timer pseudointerrupts. Pseudointerrupts, in this case, allow each of several software routines to believe that it has exclusive use of the high-resolution Programmable Interval Timer. In a master workstation, for example, the Cluster Line Protocol Handler, the 3270 terminal emulator, and a user-written device handler for real-time data acquisition equipment would concurrently need high-resolution interval timing. Each of the three pseudointerrupt handlers performs the same logical (but not device-dependent) processing as if it were servicing an external interrupt from the Programmable Interval Timer itself.

Interrupt Handlers

OS interrupt handlers are provided for each interrupt type. For interrupt types that are not expected to occur, the Extraneous Interrupt Handler calls the Carsh operation (see the "Contingency Management" section) to terminate OS operation in an orderly manner, display the termination code, and restart the OS.

Each interrupt handler services all interrupts of a single type. For example, the interrupt handler that services NMIs must accommodate all four kinds of NMIs. If another interrupt handler is substituted for the B20 NMI handler, the substitute must also handle all four kinds of NMIs.

The OS supports two kinds of interrupt handlers: mediated and raw.

Communications Interrupt Handlers

Because both SIO communications channels are served by type 9 interrupts, the communications interrupt handler should not be replaced unless both channels are to be controlled by the user-written interrupt handler. The communications interrupt handler is required in a cluster workstation, in the master workstation of a minicluster configuration, or when a communications program, such as the Asynchronous Terminal Emulator utility or the 3270 terminal emulator, is to be used.

The communications interrupt handler that services type 9 interrupts determines which of the two communications channels caused the interrupt and dispatches to the appropriate Communication Interrupt Service Routine. (See "Communications Interrupt Service Routines".)

Packaging of Interrupt Handlers

Additional interrupt handlers can be linked either to a task of an application system or to a system service process. The system service process can be linked to the System Image at system build or dynamically installed.

Application System. Packaging an interrupt handler with an application system permits the interrupt handler to occupy memory only when the application system that needs it is in memory. Also, it requires somewhat less effort to package the interrupt handler with an application system. An interrupt handler that is used by only one application system and not by others should generally be packaged with the application system.

The SetIntHandler operation is used to inform the OS of the existence of an interrupt handler in an application system.

System Service Process. If an interrupt handler must be available continuously, even while one application system is being replaced with another, then the interrupt handler must be packaged with a system service. An interrupt handler that supports a device attached to a master workstation on behalf of application systems executing in cluster workstations must be packaged with a system service in the master workstation (and must also use the formal Request/Respond model of interprocess communication). Packaging an interrupt handler with a system service reduces the size of the run files of the application systems that would otherwise include the interrupt handler. An interrupt handler that is used by all or most application systems should generally be packaged with a system service.

The SetIntHandler operation is used to inform the OS of the existence of an interrupt handler in a dynamically installed system service.

Mediated Interrupt Handlers

A mediated interrupt handler (MIH) is easier to write than raw interrupt handlers (it can be written in FORTRAN or Pascal, as well as assembly language), permits automatic nesting by priority since processor interrupts are enabled during its execution, and can communicate its results to processes through the PSend and Send operations. (See the "Interprocess Communication Management" section.) MIHs are recommended except where specifically contraindicated.

For an MIH, the entry in the Interrupt Vector Table points to a procedure in the Kernel that:

o   saves the hardware context on to the stack that is active at the time of the interrupt,

o   switches the stack (SS:SP) to a special stack that is reserved for the use of MIHs,

o   enables processor interrupts (turns on the processor interrupt-enable flag),

o   establishes the data segment appropriate to the MIH, and

o   calls the MIH at the memory address (CS:IP) of its entry point.

The MIH is responsible for giving an End-Of-Interrupt (EOI) command to the slave 8259A PIC, if any, on the Multibus board that caused the interrupt. However, it <u>must</u> <u>not</u> give an EOI command to the master 8259A PIC.

The only operations an MIH can use are PSend, Send, SetTimerInt, and ResetTimerInt. (The first two operations are described in the "Interprocess Communication Management" section; the latter two in the "Timer Management" section.)

After it completes its processing, the MIH returns to the Kernel by using a RET (not IRET) instruction.

Upon return from the MIH, the Kernel issues a nonspecific EOI command to the master 8259A PIC if the interrupt was caused by an external maskable interrupt (that is, was caused by the 8259A PIC).

If interrupts are nested and a lower priority interrupt handler was interrupted, the Kernel unconditionally returns control to the point-of-interrupt (within the lower priority interrupt handler). If the MIH sent a message to a higher priority process than the one executing at the time of the interrupt, the Kernel establishes the context of and returns control to the higher priority process. Otherwise, the Kernel reestablishes the context of and returns control to the interrupted process.

A <u>raw</u> <u>interrupt</u> <u>handler</u> (RIH) provides the fastest execution since the entry in the Interrupt Vector Table points directly to the entry point of the RIH.

An RIH is useful for servicing a high-speed non-DMA device that causes an interrupt whenever a byte is to be transferred. To service such a device, the RIH saves the minimum number of registers, transfers the byte, issues an EOI command to the master 8259A PIC (and slave 8259A PIC, if appropriate), restores the saved registers, and uses the IRET instruction to reenable processor interrupts while returning to the point of interrupt.

When the RIH determines (through counting bytes or examining the bytes being transferred) that a complete logical block was transferred, it converts itself to a mediated interrupt handler (using the MediateIntHandler operation). It then uses the PSend or Send operation to inform the device handler (or other) process that the block was transferred, issues an EOI command, if appropriate, to a slave (but not to the master) 8259A PIC, and uses a RET (not IRET) instruction to transfer control to the Kernel. The Kernel then performs the conventional termination sequence for a mediated interrupt handler. This includes issuing a nonspecific EOI command to the master 8259A PIC (but not to a slave 8259A PIC, if any).

An RIH uses the stack of the process it interrupted. It is responsible for saving and restoring all registers it uses and for giving an EOI command to the master 8259A PIC (and slave 8259A PIC, if appropriate). An RIH must leave processor interrupts disabled. Because an RIH cannot be interrupted, nesting of interrupts cannot occur while an RIH is executing. An RIH can serve an internal or external interrupt, but not a pseudointerrupt.

The only operation an RIH can use is MediateIntHandler. The MediateIntHandler operation permits an RIH to be converted to an MIH during interrupt processing.

Communications Interrupt Service Routines (CISR) are similar to MIHs except that a CISR serves only one of the two communications channels of the SIO communications controller.

CISRs can be linked to the System Image and declared at system build. Alternatively, they can be linked to a dynamically installed system service or an application system and declared through the use of the SetCommISR operation.

CISRs differ from MIHs in that the communications channel number (0 or 1) is passed to the CISR as a parameter. The CISR must have one parameter specified in its procedure definition. For an assembly language program, this means that the return is by means of an intersegment RET 2 instruction.

CAUTION: Read the "Communications Management" section of this Manual and the "Communications (SIO) Programming" section of the B20 System Programmers and Assembler Reference Manual (Part 1), form 1148699, to understand the responsibilities of a communications interrupt handler with regard to the use of the LockIn and LockOut operations, preservation of the "status affects vector" mode when programming Channel B of the SIO communications controller, and other critical issues.

Printer Interrupt Service Routines

Printer Interrupt Service Routines (PISR) are similar to MIHs except that a PISR serves only one of several devices connected to the 8259A PIC level 4 (interrupt type 12) interrupts. A PISR serves parallel printer interrupts without also servicing keyboard, Real-Time Clock, and other level 4 interrupts.

PISRs can be linked to the System Image and declared at system build. Alternatively, they can be linked to a dynamically installed system service or an application system and declared through the use of the SetLpISR operation.

OPERATIONS: PRIMITIVES AND SERVICES

Interrupt handlers provides the operations listed below.

MediateIntHandler

converts a raw interrupt handler to a mediated interrupt handler.

ResetCommISR

purges the CISRs previously established for the specified communications channel.

SetCommISR

establishes the CISRs for the specified communications channel.

SetIntHandler

establishes a raw or mediated interrupt handler.

SetLpISR

establishes the PISR to process interrupts generated by the parallel printer interface.

MediateIntHandler

Description

The MediateIntHandler primitive converts a raw interrupt handler to a mediated interrupt handler. Before using MediateIntHandler, the raw interrupt handler must save the contents of the registers of the interrupted process on the stack of the interrupted process in the following order: AX, BX, DS, CX, ES, SI, DX, DI, BP.

After that, the argument to the call should be pushed on to the stack. No other information can be pushed on to the stack at the time of the call to MediateIntHandler. MediateIntHandler switches the stack (SS:SP) to point to the MIH stack.

Procedural Interface

MediateIntHandler (fDeviceInt): ErcType

where

fDeviceInt   is TRUE or FALSE. TRUE (0FFh) indicates the interrupt handler serves device-generated interrupts. FALSE (0) indicates the interrupt handler serves software-generated interrupts (traps).

Request Block

MediateIntHandler is a Kernel primitive.

ResetCommISR

Description

The ResetCommISR service purges the CISRs
previously established for the specified
communications channel. Future interrupts from
the specified channel are ignored.

Procedural Interface

ResetCommISR (iLine): ErcType

iLine   is SIO Channel A if iLine is 0 and
SIO Channel B if iLine is 1.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	2
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	102
12	iLine	2	

SetCommISR

Description

The SetCommISR service establishes the CISRs for the specified communications channel. Separate CISRs are established to process transmit, external/status, receive, and special receive conditions.

Procedural Interface

SetCommISR (iLine, pDS, pTxIsr, pExtIsr, pRxIsr,
           pSpRxIsr): ErcType

where

iLine   is SIO Channel A if it is 0 and SIO Channel B if it is 1.

pDS   is the memory address of any byte in the memory segment to be used as the data segment of the CISRs. The segment base address part of pDS is to be used as the data segment base (that is, loaded into the DS register) when any of the four CISRs is activated.

pTxIsr   is the memory address (CS:IP) of the CISR that is to process Transmit-Data-Buffer-Empty interrupts.

pExtIsr   is the memory address (CS:IP) of the CISR that is to process External/Status interrupts.

pRxIsr   is the memory address (CS:IP) of the CISR that is to process Receive-Character-Available interrupts.

pSpRxIsr   is the memory address (CS:IP) of the CISR that is to process Receive-Special-Condition interrupts.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	22
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	101
12	iLine	2	
14	pDS	4	
18	pTxIsr	4	
22	pExtIsr	4	
26	pRxIsr	4	
30	pSpRxIsr	4	

SetIntHandler

Description

The SetIntHandler service establishes a raw or mediated interrupt handler. When an application system terminates, its interrupt handler is detached and the default interrupt handler again serves the interrupts.

Procedural Interface

SetIntHandler (iInt, pIntHandler, saData,
             fDeviceInt, fRaw): ErcType

where

iInt   is the interrupt type (0-119).

pIntHandler   is the entry point of the interrupt handler.

saData   is the segment base address of the data segment that is used by the mediated interrupt handler (for mediated interrupt handlers only).

fDeviceInt   is TRUE or FALSE. TRUE (0FFh) indicates the interrupt handler serves device-generated interrupts. FALSE (0) indicates the interrupt handler serves software-generated interrupts (for mediated interrupt handlers only).

fRaw   is TRUE or FALSE. TRUE (0FFh) indicates the interrupt handler serves raw interrupts. FALSE (0) indicates the interrupt handler serves mediated interrupts.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	12
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	69
12	iInt	2	
14	pIntHandler	4	
18	saData	2	
20	fDeviceInt	2	
22	fRaw	2	

SetLpISR

Description

> The SetLpISR service establishes the PISR to process interrupts generated by the parallel printer interface. A PISR established by an application system is reset automatically when the application system terminates.

Procedural Interface

> SetLpISR (pLpIsr, saData): ErcType
>
> where
>
> pLpIsr   is the memory address (CS:IP) of the printer interrupt handler. If it is 0, it resets the current interrupt handler.
>
> saData   is the value of the data segment (DS) which is used by the printer interrupt handler.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	121
12	saData	2	
14	pLpIsr	4	

SECTION 30
CONTINGENCY MANAGEMENT

OVERVIEW

Contingency refers to a variety of hardware and software conditions that have undesirable effects. These conditions can be hardware faults such as a memory parity error, OS-detected inconsistencies such as a bad checksum of a Volume Home Block, or application system-detected conditions.

The OS always terminates execution when it detects an inconsistency. The default handling of hardware faults (nonmaskable interrupts) is to terminate system operation; however, nonmaskable (type 2) interrupts can be directed to a user-written interrupt handler linked to the System Image or declared through the use of the SetIntHandler operation (see the "Interrupt Handlers" section).

OS crash conditions are logged in the Log File ([Sys]<Sys>Log.Sys). The OS also logs disk controller faults, disk input/output errors, and fatal communications errors in the cluster configuration there. The application system can also use the WriteLog operation to write records to the Log File. The PLog utility prints the Log File (see the B20 System Software Operation Guide, form 1148772). (The Log File is also discussed in the "File Management" section.)

OPERATIONS: PROCEDURES AND SERVICES

Contingency management provides the operations listed below.

Crash   causes OS operation to
                         terminate, a crash dump to be
                         written, the OS to be reloaded,
                         and the Executive to display
                         the cause of the crash when it
                         is restarted.

FatalError   terminates operation of the
                         application system after a
                         catastrophic event.

WriteLog   writes a variable-length record
                         to the Log File.

Crash

Description

The Crash procedure causes OS operation to terminate, a crash dump to be written to the file [Sys]<Sys>CrashDump.Sys, the OS to be reloaded, and the Executive to display the cause of the crash when it is restarted. Crash never returns.

Procedural Interface

Call Crash (ercTermination)

where

ercTermination
is a 16-bit status code to be displayed by the Executive after the OS is reloaded.

Request Block

Crash is a system common procedure.

FatalError

Description

The FatalError procedure terminates operation of the application system after a catastrophic event. The Burroughs-supplied version of FatalError consists of a call to the ErrorExit operation (see the "Task Management" section).

Burroughs object module procedures call FatalError rather than the ErrorExit operation when they encounter inconsistencies. This allows the system builder to easily substitute a user-written version of FatalError that does one of the following:

o   invokes the Debugger,

o   calls the Crash operation because it causes a crash dump to be written that is useful in debugging,

o   calls the Crash operation because the application system wants to terminate when it malfunctions, or

o   provides special logic to attempt an orderly system shutdown when the application system detects a malfunction. Such code is best included in a user-written version of FatalError.

Procedural Interface

Call FatalError (ercTermination)

where

ercTermination
is a 16-bit status code to be placed in the Application System Control Block for interrogation by the Executive. A nonzero status code causes the contents of the type-ahead buffer to be discarded and the submit or recording file to be closed.

Request Block

FatalError is an object module procedure.

WriteLog

Description

The WriteLog service is used by an application
system to write a variable-length record to the
Log File ([Sys]<Sys>Log.Sys). The PLog
utility (see the B 20 System Software Operation
Guide, form 1148772) prints the Log File. The
PLog utility interprets the first word of the
record as an error type; the reset of the record
is not interpreted. When the record is written
to the Log File, additional information is
inserted by the BTOS Operating System. The size
of the entry in the Log File is the sum of the
size of the record and the size of the
additional information.

Procedural Interface

WriteLog (pbRecord, cbRecord): ErcType

where

pbRecord
cbRecord   describe the record to be logged.
                Its maximum size is 255 bytes.

Request Block

Offset	Field	Size (bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	125
12	reserved	6	
18	pbRecord	4	
22	cbRecord	2	

Log File Record Format

Offset	Field	Size (bytes)	Description
0	sEntry	1	The size of the entry
1	datetime	4	The time of logging, in simplified Date/Time format.
5	ErrorType	2	The error type of the entry. The error type is a 16-bit word. The defined types are as follows:
			0 ASCII character messages
			61440 to 65535 reserved by BTOS to log errors and events
7	hardwaretype	1	
8	fNoFileSystem	1	
9	ClusterConfig	1	
10	fCommIOP	1	
11	fMultiPartition	1	Information from the System Configuration Block (described in this manual) of the workstation.
12	reserved	3	
15	lineNumber	1	The line number of the workstation making the log, if it is a CommIOP-Cluster workstation.

Offset	Field	Size (bytes)	Description
16	idNumber	1	The cluster ID of the workstation if it is cluster workstation.
17	sMemory	2	The amount of memory in the workstation.
19	userName	12	The first 12 characters of the user name at the time of the error.
31	typeSpecificInfo		Information supplied by caller of WriteLog.

The ErrorType and typeSpecificInfo fields are supplied by the caller of WriteLog, the rest of the information is supplied by BTOS.

APPENDIX A
STATUS CODES

Codes marked with an asterisk (*) cause OS termination and an automatic reload.

	Decimal Value	Hexa-decimal Value	Meaning
General			
	0	0000	OK. Successful completion.
	1	0001	End of file (EOF).
	2	0002	End of medium (EOM). An attempt to read or write beyond the end of a file or device.
	3*	0003	Inconsistency. Run the crash dump analyzer.
	4	0004	Operator intervention.
	5	0005	Syntax error.
	6	0006	Master workstation not running. Interstation communication with the master workstation of the cluster has been interrupted.
	7	0007	The procedures necessary to implement this operation were excluded at system build.
	8	0008	An internal inconsistent state is discovered.
Kernel			
	10	000A	Exchange out of range.
	11	000B	Bad pointer.
	12	000C	No link block. Generated by PSend. (See the "Interprocess Communication Management" section.)

Decimal Value	Hexa-decimal Value	Meaning
13	000D	Bad interrupt vector. Generated by SetIntHandler. (See the "Interrupt Handlers" section.)
14	000E	No message available.
15	000F	No link block available. Generated by Send and Request. (See the "Interprocess Communication Management" section.)
16	0010	Inconsistent request block.
17	0011	Mismatched respond.
18	0012	No PCB available. Create fewer processes or specify more Process Control Blocks at system build.
19*	0013	PIT chain bad. Programmable Interval Timer block that was established by SetTimerInt was erroneously modified. (See the "Timer Management" section.)
20	0014	Invalid response exchange specified in request block.
21*	0015	Memory protect violation.
22*	0016	Bus time out. Attempted access to a nonexistent memory location or input/output port.
23*	0017	Memory parity failure.
24*	0018	Power failure.
25*	0019	Unknown NMI.
26*	001A	Stray interrupt. Interrupt of unexpected interrupt type.

Decimal Value	Hexa-decimal Value	Meaning
27*	001B	Divide error.
30*	001E	Request table inconsistent.
31	001F	No such request code.
32	0020	Bad message on default response exchange.
33	0021	Service not available. The request is not ready to be served by the system service process. The installed system service process has to call ServeRq to declare its readiness to service the specified request code.
34	0022	Exit run file is not specified.
35	0023	Wrong Overlay: Wrong file system overlay has been loaded. Overlays specified in ObjLink file must be in correct order at system build.

Cluster Request Management

Decimal Value	Hexa-decimal Value	Meaning
40*	0028	Not enough cluster buffer memory. Initialization error in master workstation. Insufficient memory is available to allocate for cluster buffers. Specify smaller data structures at system build.
41	0029	No available RCB. No RCB is available at the local cluster workstation Agent Service Process to process this request. Specify more RCBs at system build or modify the application system to require fewer concurrent requests.
42	002A	Agent Srp no room. User-defined request block is too big for the Agent to handle.
43*	002B	Bad response from master workstation. The response from the master workstation does not match the request.

Decimal Value	Hexa-decimal Value	Meaning
44*	002C	Unmatched response at master workstation Agent Service Process. Probably a message was erroneously sent to exchange 12 at the master workstation.
45	002D	Request block too big. The request block (with data fields expanded) is too big for the transmission buffer or line buffer. Reduce the size of the request or specify larger buffers at system build.
46	002E	Master workstation going down. Polling of the cluster workstation is going to stop.

Initialization

100*	0064	Memory failure detected during initialization.
101*	0065	Insufficient memory for OS initialization.
102*	0066	No DCB for the device from which the OS was bootstrapped.
103	0067	Initialization error. The Operating System logs this (see PLog in form 1148772.) during initialization if it finds something wrong with the keyboard, video display, and so forth.

File Management

201	00C9	No free volume structure. The Volume Home Block and Device Control Block values specified at system build are inconsistent.

Decimal Value	Hexa-decimal Value	Meaning
202	00CA	Directory full. Rename all the files in this directory to another directory and then delete this directory. Create a new directory with the name of the old directory. Then rename all the files from the other directory to this new, expanded directory.
203	00CB	No such file.
204	00CC	No such directory.
205	00CD	Bad file specification.
206	00CE	Bad user number. The user number should always be 0.
207	00CF	Bad request code.
208	00D0	Duplicate volume.
209	00D1	File is read only.
210	00D2	Bad file handle.
211	00D3	Bad buffer size. This must be a multiple of 512 for disk volumes.
212	00D4	Bad logical file address. This must be a multiple of 512 for disk volumes.
213	00D5	No free FAB. Open fewer files concurrently or specify more File Area Blocks at system build.
214	00D6	No free file number. Open fewer files concurrently or specify more File Control Blocks per User Control Block at system build.

Decimal Value	Hexa-decimal Value	Meaning
215	00D7	No such volume or no such device. The volume is currently not mounted.
216	00D8	Volume not mounted.
217	00D9	Bad password.
218	00DA	Bad mode.
219	00DB	Access denied: A file system request was made which was denied because of wrong password or because of an illegal request to modify or access a system file (such as <Sys>Sysimage.Sys).
220	00DC	File in use. A process that opens a file in modify mode is guaranteed exclusive access. Only one file handle can refer to a file that is open in modify mode.
221	00DD	File Header bad checksum. The volume control structures are invalid. Run Backup Volume, IVolume, and Restore on this volume.
222	00DE	File Header bad page number. The volume control structures are invalid. Run Backup Volume, IVolume, and Restore on this volume.
223	00DF	File Header bad header number. The volume control structures are invalid. Run Backup Volume, IVolume, and Restore on this volume.
224	00E0	File already exists.
225	00E1	No free File Headers. Run Backup Volume, IVolume (and specify more File Header Blocks), and Restore on this volume.

Decimal Value	Hexa-decimal Value	Meaning
226	00E2	Free File Headers broken. The volume control structures are invalid. Run Backup Volume, IVolume, and Restore on this volume.
227	00E3	Device in use.
228	00E4	Device already mounted.
229	00E5	Device not mounted.
230	00E6	Disk full. There are not enough available disk sectors to accommodate the current CreateFile or ChangeFileLength request.
231	00E7	Not a device that can be mounted.
232	00E8	No valid VHB. The volume control structures are invalid. Run Backup Volume, IVolume, and Restore on this volume.
233	00E9	File Header bad file name. The volume control structures are invalid. Run Backup Volume, IVolume, and Restore on this volume.
234	00EA	Odd byte buffer address. The buffer must be word-aligned.
235	00EB	Wrong volume mounted.
236	00EC	Bad device specification.
237	00ED	Directory page invalid. The volume control structures are invalid. Run Backup Volume, IVolume, and Restore on this volume.
238	00EE	Request not valid for device.

Decimal Value	Hexa-decimal Value	Meaning
239	00EF	Wrong volume destination. Rename cannot move a file to another volume.
240	00F0	Directory already exists.
241	00F1	Directory not empty.
242	00F2	MFD is full. Run Backup Volume, IVolume (and specify more sectors for the Master File Directory), and Restore on this volume.
243	00F3	Verify error. A volume control structure (VHB, FHB, etc.) was written and then immediately reread to verify that it was written correctly. The information reread does not compare with the information written, although the disk controller did not report an error. Error 243 indicates a serious disk controller, DMA, or memory hardware malfunction.
244*	00F4	System device not ready. If a swapping OS was bootstrapped from a floppy disk, then the OS floppy disk cannot be removed from the drive.
245	00F5	Run file bad checksum. The file is probably not a run file.
246	00F6	Bad run file. The file is probably not a run file.
247	00F7	Old format run file. The file is probably not a run file.

Decimal Value	Hexa- decimal Value	Meaning
248	00F8	Wrong pRq argument. CheckReadAsync or CheckWriteAsync does not agree with the preceding ReadAsync or WriteAsync.
249	00F9	Invalid attributes for secondary task. A task loaded with LoadTask (as opposed to Chain) cannot use virtual code segments or have a memory array. (See form 1148681 and the "Task Management" section.)
250	00FA	Too many runs. The file cannot be expanded because it already contains the maximum number of runs. The maximum number of runs per file is a system build parameter.
251	00FB	Cannot write to the [Sys]<Sys>Log.
252	00FC	Cannot open the OS image file for the swapping cluster workstation.
253	00FD	Cannot read the OS overlay for the swapping cluster workstation.
254	00FE	All the user numbers on the master workstation have been used. Change the system build parameter for the User Control Block (multiple application partitions only).
290	0122	Log buffer overflow. Multiple errors occurred rapidly and the Operating System was unable to log all of them.

	Decimal Value	Hexa-decimal Value	Meaning

Device Management

	Decimal Value	Hexa-decimal Value	Meaning
	300	012C	Device not ready. Make sure that the power is on and that there is a floppy disk in the disk drive.
	301	012D	I/O error. If you are using a floppy disk, the disk is bad and should be replaced with another disk.
	302	012E	Write protected. There is no write enable tab on the floppy disk.
	303	012F	No free IOB. There are too many concurrent input/output operations. More I/O Blocks should be specified at system build.
	304	0130	Odd DMA Count: The number of bytes transferred by DMA must be even.

Floppy Disk Controller

These codes relate to hardware controller failure. Code 328 (decimal) may result ´from a failure to include all floppy disks in the system build.

	Decimal Value	Hexa-decimal Value	Meaning
	320	0140	Floppy disk controller busy in command.
	321	0141	Floppy disk controller never ready in command.
	322	0142	Floppy disk controller data input in command.
	323	0143	Floppy disk controller never ready in result.
	324	0144	Floppy disk controller not data input in result.
	325	0145	Floppy disk controller not busy after Xfer request.

Decimal Value	Hexa-decimal Value	Meaning
326	0146	Floppy disk controller wrong unit after Xfer request.
327	0147	Floppy disk controller busy without Xfer request.
328	0148	Floppy disk controller interrupt from undefined unit.
329	0149	Floppy time out.
330	014A	Incomplete DMA transfer to/from floppy disk.

Hard Disk Controller

These codes relate to hardware controller failure. Code 348 (decimal) may result from a failure to include all Winchester disks in the system build.

340	0154	Hard disk controller busy in command.
341	0155	Hard disk controller never ready in command.
342	0156	Hard disk controller data input in command.
343	0157	Hard disk controller never ready in result.
344	0158	Hard disk controller not data input in result.
345	0159	Hard disk controller not busy after Xfer request.
346	015A	Hard disk controller wrong unit after Xfer request.
347	015B	Hard disk controller busy without Xfer request.
348	015C	Hard disk controller interrupt from undefined unit.

Decimal Value	Hexa-decimal Value	Meaning
349	015D	Hard disk time out.
350	015E	Incomplete DMA transfer to/from hard disk.
351	015F	Bad hard disk controller.

Allocation

Decimal Value	Hexa-decimal Value	Meaning
400	0190	Not enough memory available to satisfy memory allocation request.
401	0191	Cannot allocate long-lived memory. The memory cannot be allocated becuase the Debugger is locked into memory in multiple-process or interrupt mode. (See form 1148665.)
402	0192	Invalid memory segment specification to DeallocMemorySL/LL. (See the "Memory Management" section.)
410	019A	All exchanges already allocated. Specify more exchanges at system build. Also caused by too many files listed in the Submit command. Submit fewer files at a time.
411	019B	Invalid exchange identification specified to DeallocExch. (See the "Exchange Management" section.)

Timer Management

Decimal Value	Hexa-decimal Value	Meaning
420	01A4	Too many RTC requests. Specify a bigger Real-Time Clock request table at system build.
421	01A5	Invalid timer block specification in CloseRTClock.

	Decimal Value	Hexa-decimal Value	Meaning
Task Management			
	430*	01AE	Cannot load Executive. The Debugger is locked in memory, [Sys]<Sys>Exec.Run is bad, or the memory specifications at system build were erroneous.
	431	01AF	The printer ISR already exists.
Video Display Manager			
	500	01F4	Frame number/coordinates do not agree with the VCB.
	501	01F5	Invalid argument to VDM.
	502	01F6	Video buffer is not word-aligned.
	503	01F7	VCB not completely initialized.
	504	01F8	Video DMA hardware failure.
	505	01F9	Too many attributes on a line (B 21 System only).
Keyboard Management			
	601	0259	Duplicate ReadKbd or ReadKbdDirect. Only one ReadKbd or ReadKbdDirect request can be outstanding at a time.
	602	025A	No character available. ReadKbdDirect specified not to wait for a character and no keyboard character/code is currently available.
	603	025B	Invalid escape sequence in submit file.

Decimal Value	Hexa-decimal Value	Meaning
604	025C	Invalid argument to a keyboard operation.
605	025D	Invalid mode code to SetSysInMode.
606	025E	Failure of 8048 keyboard microprocessor.
607	025F	Reserved.
608	0260	Application system being terminated by request of another process or ACTION-FINISH.
609	0261	No action code available. ReadActionCode returns this status if the workstation operator has not entered an action code.
610	0262	Type-ahead buffer overflow.

Printer Spooler

Decimal Value	Hexa-decimal Value	Meaning
700	02BC	A ConfigureSpooler operation attempted to free a printer that was not attached.
701	02BD	A SpoolerPassword operation attempted to enter a password when the printer spooler was not waiting for a password.
702	02BE	Invalid printer name specified in a SpoolerPassword operation.
703	02BF	Invalid channel number specified in a ConfigureSpooler operation.
704	02C0	A ConfigureSpooler operation attempted to add a new printer to a channel that is not free
705	02C1	Invalid printer spooler configuration file specified in a ConfigureSpooler operation.
706	02C2	A spooler was installed with a printer name which was already in use. Printer names must be unique.

Decimal Value	Hexa-decimal Value	Meaning
707	02C3	BadPrinter Configuration File: The "cbConfigureFile" field of the ConfigureSpooler service exceeds 91 characters.
708	02C4	Bad Queue Name: The "cbQueueName" field of the ConfigureSpooler service exceeds 50 characters.

Application Partition Management

Decimal Value	Hexa-decimal Value	Meaning
800	0302	Application partition is not vacant. Vacate the partition first.
801	0303	Cannot create any more application partitions. Number of application partitions is a system build parameter.
802	0304	Partition name is duplicated. Default name for the first application partition is 'bkg.
803	0305	Invalid partition handle is specified.
804	0306	Invalid partition name is specified. Indicates too many characters or illegal characters.
805	0307	Application partition is vacant.
806	0308	Application partition is locked. A task on a locked partition cannot be terminated.
807	0309	Application partition is not locked. The partition should be locked before using the SetPartitionExchange operation.
808	030A	Partition exchange has not been set.
809	030B	Partition exchange has already been set.

	Decimal Value	Hexa-decimal Value	Meaning
	810	030C	An Assign statement in JCL file is attempted when the assign table is full.
Queue Management			
	900	0384	A DeleteMarkedQueueEntry, UnmarkQueueEntry, or RewriteMarkedQueueEntry operation was invoked with an invalid queue entry handle (qeh). The qeh specified was for an entry that is not marked.
	901	0385	A DeleteKeyedQueueEntry operation specified an entry that was previously marked.
	902	0386	A DeleteKeyedQueueEntry, ReadKeyedQueueEntry, or MarkKeyedQueueEntry operation was invoked for which no matching entry was found.
	903	0387	A MarkNextQueueEntry operation was invoked when no entries were available.
	904	0388	The ReadNextQueueEntry operation specified an entry that was deleted since its queue entry handle was returned.
	905	0389	The pb/cbQueueName fields of an operation specifies an invalid queue.
	906	038A	An EstablishQueueEntry operation was invoked when 100 server processes were already established.
	907	038B	A Marking operation was invoked by a server process that had not invoked an EstablishQueueServer operation.

Decimal Value	Hexa-decimal Value	Meaning
908	038C	An AddQueueEntry operation was involved with the fQueueIfNoServers flag set to FALSE when no server processes were established.
909	038D	A DeleteMarkedQueueEntry, UnmarkQueueEntry, or RewriteMarkedQueueEntry operation was invoked with an invalid queue entry handle.
910	038E	A DeleteMarkedQueueEntry, UnmarkQueueEntry, or RewriteMarkedQueueEntry operation was invoked by a server process other than the server process that marked the entry.
911	038F	A syntax error was made when opening the queue index file.
912	0390	An AddQueueEntry operation specifies a queue type that does not match the queue type in the queue index file.
913	0391	An AddQueueEntry operation was invoked with an invalid date/time specification.
914	0392	The server process specified in an EstablishQueueServer operation is already established as a server.

Debugger

(See form 1148665 for additional information.)

1001	03E9	Cannot convert from simple mode to multiple-process mode. To enter multiple-process mode, first exit the Debugger and then press ACTION-B.

Decimal Value	Hexa-decimal Value	Meaning
1002	03EA	Not enough memory for multiple-process mode or CODE-I breakpoint. Approximately 37k of memory must be available to enter multiple-process mode or set a CODE-I breakpoint.
1003	03EB	Cannot deactivate Debugger. The Debugger cannot be deactivated while CODE-I breakpoints are set or while a breakpoint has just executed. To deactivate the Debugger, first remove all CODE-I breakpoints and/or proceed (single step) from the breakpoint.
1004	03EC	Breakpoint already there. The Debugger allows only one breakpoint per location.
1005	03ED	Debugger crash. A fatal internal error has occurred. Press the RESET button on the back of the workstation.

Sequential Access Method

Decimal Value	Hexa-decimal Value	Meaning
2305	0901	Too many put backs. Only one PutBackByte is allowed before reading again.
2315	090B	Invalid mode to OpenByteStream.
2325	0915	Invalid BSWA. BSWA has been erroneously modified by the user or a byte stream was not opened for BSWA.
2335	091F	Buffer too small. Buffer must be 1024 bytes to allow device independence.
2336	0920	Invalid video byte stream escape sequence.
2340	0924	Parity error detected on the last byte received by the communications byte stream. All bytes, except the last one, returned from the read operation were received without error.

Decimal Value	Hexa-decimal Value	Meaning
2341	0925	Receive overrun error detected on the last byte received by the communications byte stream. All bytes, except the last one, returned from the read operation were received without error.
2342	0926	Framing error detected on last byte received by the communications byte stream. All bytes, except the last one, returned from the read operation were received without error.
2343	0927	Wrong configuration type. The specified configuration file is not of the type expected for the device specified.
2344	0928	Bad configuration file. There was an error in accessing the appropriate configuration file. Either the specified configuration file (or the default if one was not specified) does not exist or an error was encountered when trying to read the file.

X.25 Sequential Access Method

Decimal Value	Hexa-decimal Value	Meaning
2350	092E	X.25 error occurred during operation. If an X.25 error occurs during a byte stream operation, the call is cleared, but the byte stream is not closed. A ReleaseByteStream or CloseByteStream operation must be done to close the byte stream.
2351	092F	Time out. The specified time out elapsed before the X.25 Network Gateway system service operation finished. The operation in question is terminated, but the call is not cleared.

Decimal Value	Hexa-decimal Value	Meaning

Parameter Management

2440	0988	No such iParam. The value of iParam supplied to RgParam is not less than CParams.
2450	0992	No such jParam. The value of jParam supplied to RgParam is not less than CSubparams (iParam).
2470	09A6	VLPB full. The operation failed because the Variable Length Parameter Block could not be extended by allocating long-lived memory.
2480	09B0	Illegal iParam. The value of iParam supplied to RgParamSetListStart or RgParamSetSimple is not less than CParams.
2490	09BA	Not in list. RgParamSetEltNext was invoked after a RgParamSetListStart or RgParamSetSimple other than RgParamSetListStart or RgParamSetEltNext.

Date/Time Conversion

2700	0A8C	Year out of range 1952-2042.
2701	0A8D	Day not valid for specified month. Must be 1 to 28/29/30/31 as appropriate.
2702	0A8E	Date and day of week disagree.
2703	0A8F	Invalid time of day specification.

Direct Access Method

3000	0BB8	DAWA in use. OpenDaFile failed because the DAWA is currently associated with another DAM file.

Decimal Value	Hexa-decimal Value	Meaning
3001	0BB9	Not readable by DAM. OpenDaFile failed because the specified file contains records that cannot be read by the DAM. For example; the file can contain variable-length records.
3002	0BBA	sRecord mismatch. OpenDaFile failed because the sRecord parameter did not match the sRecord specified when the file was created.
3003	0BBB	DAM internal error. The operation failed because an internal inconsistency was detected.
3004	0BBC	DAWA invalid. The operation failed because pDAWA specified an invalid DAWA. A DAWA is invalid if it is not recognized as a DAWA or if it is not associated with an open file.
3005	0BBD	Bad record fragment. ReadDaFragment or WriteDaFragment failed because the record fragment exceeds the record bounds.
3006	0BBE	Bad buffer mode. SetDaBufferMode failed because an invalid buffer mode was given.
3007	0BBF	Record beyond existing records. The operation failed because the specified record does not exist. This status code is equivalent to ercRecordDoesNot-Exist (code 3302) except that this code (that is, 3007) provides this additional information: the record is beyond any existing record.

	Hexa-	
Decimal	decimal	
Value	Value	Meaning

Indexed Sequential Access Method
(See form 1148723 for additional information.)

3100	OC1C	No such index. The specified key field does not exist.
3101	OC1D	Prefix not valid. The type of index specified to SetupISAMIterationPrefix is neither byte string nor character string.
3102	OC1E	Bad key length. The length of the key is inconsistent with the type of the index.
3103	OC1F	Bad ISAM handle. The ISAM handle does not identify an open ISAM data set.
3104	OC20	Bad ISAM header size. The ISAM data set cannot be opened by OpenISAM due to an inconsistency in the header of one of the files of the data set.
3105	OC21	Bad ISAM header. The ISAM data set cannot be opened by OpenISAM due to an inconsistency in the header of one of the files of thé data set.
3106	OC22	Too many indexes. The number of indexes in the data set created by CreateISAM or opened by OpenISAM is larger than the cIndexesMax parameter of InstallISAM or the number specified in [Maximum no. of indexes in any ISAM data set (default 10)] of ISAM Install.
3107	OC23	ISAM already installed. This code is generated by InstallISAM or ISAM Install if ISAM is already installed.

Decimal Value	Hexadecimal Value	Meaning
3108	0C24	Not enough DGroup memory. The memory area specified by the oDGroupMemory and sDGroupMemory parameters of InstallISAM is not large enough, or not enough short-lived memory can be allocated for DGroup memory, or the values of the parameters of InstallISAM require allocation of more than 64,435 bytes of DGroup memory.
3109	0C25	Not addressable from DS. The short-lived memory allocated by InstallISAM for DGroup memory is not addressable from the DS (data segment) register of the invoking process.
3110	0C26	No free ISAM Control Blocks. All the ISAM Control Blocks are in use. Reinstall ISAM with more ISAM Control Blocks.
3111	0C27	No free ISAM User Blocks. All the ISAM User Blocks are in use. Reinstall ISAM with more ISAM User Blocks.
3112	0C28	No free ISAM Index Specification Blocks. All the ISAM Index Specification Blocks are in use. Reinstall ISAM with more ISAM Index Specification Blocks.
3113	0C29	Buffers too large. The amount of memory required by the buffer sizes specified to InstallISAM or to ISAM Install exceeds one megabyte.
3114	0C2A	Bad pCacheBuffers. The relative address part of the pCacheBuffers parameter of InstallISAM is nonzero.

Decimal Value	Hexa-decimal Value	Meaning
3115	0C2B	ISAM crashed. This code is generated by all ISAM operations upon detection of an internal error.
3116	0C2C	ISAM not installed. This code is generated by all ISAM operations before InstallISAM is called or ISAM Install is successfully executed.
3117	0C2D	Bad unique record identifier. An incorrect unique record identifier parameter was passed to ISAM.
3118	0C2E	Duplicate key. A StoreISAMRecord or ModifyISAMRecord was attempted with the value of a key field that duplicates the field in another record.
3119	0C2F	Index file error. This is returned as the status code of an ISAM operation. The detail status code refers to a problem with the index file of the ISAM data set.
3120	0C30	Attempted privacy breach. An attempt was made to modify a data set which is open in batch read or transaction read mode.
3121	0C31	Bad ISAM request. The parameters of an ISAM operation are inconsistent or have invalid values.
3122	0C32	Data store file error. This is returned as the status code of an ISAM operation. The detail status code refers to a problem with the data store file of the ISAM data set.

Decimal Value	Hexa-decimal Value	Meaning
3123	0C33	Index to data error. An inconsistency has arisen between the index and data store files of the ISAM data set.
3124	0C34	Record size incorrect. The record size specified is incorrect for the ISAM data set.
3125	0C35	Duplicates allowed. An attempt was made to use ReadUniqueISAMRecord for keys for which duplicates are allowed.
3126	0C36	No such record. An attempt was made to use ReadUniqueISAMRecord to read a record that does not exist.
3127	0C37	No more records. An attempt was made by ReadNextISAMRecord or GetISAMRecords to read more records than those specified to SetupISAMIterationKey, SetupISAMIterationPrefix, or SetupISAMIterationRange.
3128	0C38	Bad key. A key does not correspond to the index type. (For example, each digit of a BCD key must be between 0 and 9.)
3129	0C39	Bad index. The specified key field does not exist.
3130	0C3A	Bad ISAM mode. OpenISAM detects a bad mode.
3131	0C3B	Cannot open ISAM. This is returned as the status code of an ISAM operation. The detail status code gives the reason for the failure.

Decimal Value	Hexa-decimal Value	Meaning
3132	0C3C	Bad ISAM password. The password does not give the access desired by OpenISAM, or the password is larger than the 12 bytes accepted by SetISAMPasswords.
3133	0C3D	Wrong size record. OpenISAM detects the wrong size record.
3134	0C3E	Incompatible ISAM mode. An attempt was made to open a data set when the data set is already open in an incompatible mode.
3135	0C3F	ISAM already locked. LockISAM was invoked while an ISAM data set is locked by the user.
3136	0C40	Not administrator. An operation for which the data set must be open in administrator mode was attempted with the data set open in some other mode.
3137	0C41	Cannot create ISAM. This is returned as the status code of CreateISAM. The detail status code gives the reason for the failure.
3138	0C42	ISAM buffer too small. The data set being opened or created requires buffers larger than those installed.
3139	0C43	Not locked. An attempt was made in transaction mode to call an ISAM operation other than CloseISAM for a data set that is not locked.
3140	0C44	Small ISAM Record. An attempt was made to create an ISAM data set with records shorter than four bytes.

Decimal Value	Hexa-decimal Value	Meaning
3141	0C45	Not in transaction. An operation was invoked for which the user must be in a transaction, but the user was not in a transaction.
3142	0C46	Data set not available. An attempt was made to read or hold a record, or to hold a data set, without queueing, and the data set was held by another user.
3143	0C47	Record not available. An attempt was made to read or hold a record without queueing, and the record was held by another user.
3144	0C48	Record not held. An operation for which the record (or its data set) must be held was invoked when neither the record nor its data set was held.
3145	0C49	Too many records held. An attempt was made to hold a record when the maximum allowable number of records are already held.
3146	0C4A	In transaction. StartISAMTransaction was invoked during a transaction.
3147	0C4B	Request pending. A transaction operation other than PurgeISAMTransaction was invoked when one or more requests were queued for the user.
3148	0C4C	Transaction purged. PurgeISAMTransaction was invoked while the request was queued.

| | Hexa- | |
| Decimal | decimal | |
Value	Value	Meaning
3149	OC4D	Not enough buffers. InstallISAM (or the ISAM Install command) was invoked specifying fewer than two data store buffers or three index buffers. Specify at least two data store buffers and three index buffers.
3170- 3199	OC62- OC7F	Internal ISAM errors.

Key-in-Record

(See forms 1148764 and 1148723 for additional information.)

3200	OC80	Bad key type. The type field of a key specification for Sort/Merge or ISAM is invalid.
3201	OC81	Incorrect key length. The cbKey field of a key specification for a Sort/Merge or ISAM operation does not correspond to the type field of the key specification. (For example, for binary keys, cbKey must be 2.)
3202	OC82	Bad key. A key contained in a record for Sort/Merge or ISAM, or a key described by a parameter of an ISAM operation, is not of the correct type. (For example, each digit of a BCD key must be between 0 and 9.)

Standard Access Methods

3300	OCE4	Not a STAM file. The operation failed because the file did not contain the proper signature.
3301	OCE5	STAM header bad checksum. The operation failed because the checksum computed on the file header did not match the checksum computed when the file was created.
3302	OCE6	Record does not exist. The operation failed because the specified record does not exist.

Decimal Value	Hexa-decimal Value	Meaning
3303	0CE7	Malformed record. The operation failed because data read from the disk contained an inconsistency in the record header and trailer.
3304	0CE8	Not fixed-length record. The operation failed because the access method cannot reference variable-length records.
3305	0CE9	Bad file type. The operation failed because the file cannot be accessed with the specified access method.
3306	0CEA	Bad buffer size. The operation failed because the buffer size was too small or not a multiple of 512.
3307	0CEB	Buffer not word-aligned. The operation failed because the buffer was not word-aligned.

External-Key Sort

(See form 1148764 for additional information.)

Decimal Value	Hexa-decimal Value	Meaning
3400	0D48	Cannot open work file. Unable to open one of the work files during PrepareSort.
3401	0D49	Work area bad. Unable to allocate work area during PrepareSort.
3402	0D4A	Bad key size. A key passed to ReleaseRecordAndKey is a different length than the length specified in PrepareSort.
3403	0D4B	File error during sort. A file error occurred during the sort phase of the program.

Decimal Value	Hexa-decimal Value	Meaning
3404	0D4C	No more records. ReturnRecordAndKey was called after all records were retrieved.
3405	0D4D	Error returning record. An error occurred in ReturnRecordAndKey.
3406	0D4E	Error during conclude. An error occurred in ConcludeSort or TerminateSort.
3407	0D4F	More records available. ConcludeSort was called before all records were retrieved. To end a sort prematurely, call TerminateSort.
3408	0D50	Record too large. The size of a record is larger than the maximum key size specified in PrepareSort or the sort area is not large enough.
3409	0D51	Error during sort. An error occurred during DoSort.
3410	0D52	Insufficient memory. Not enough memory was allocated for the sort work area.
3411	0D53	No records to sort. DoSort was called before any records were released.

Key-in-Record Sort

(See form 1148715 for additional information.)

Decimal Value	Hexa-decimal Value	Meaning
3500	0DAC	Sort pending. PrepareKeySort was called while a sort was already active.
3501	0DAD	No sort pending. A sort procedure other than PrepareKeySort was called before PrepareKeySort.

Decimal Value	Hexa-decimal Value	Meaning
3502	0DAE	Bad sort key. The key provided is inconsistent with its specifications.
3503	0DAF	Sort key not in record. A key could not be synthesized from this record, given the initial specifications of keys within records.
3504	0DB0	Bad key specification. The key specification in PrepareKeySort is incorrect. It conflicts with the maximum record size provided.

Record Sequential Access Method

Decimal Value	Hexa-decimal Value	Meaning
3600	0E10	RSWA in use. OpenRsFile failed because the RSWA is currently associated with another RSAM file.
3601	0E11	RSWA invalid. The operation failed because pRSWA specified an invalid RSWA. An RSWA is invalid if it is not recognized as an RSWA or if it is not associated with an open file.
3602	0E12	RSAM internal error. The operation failed because an internal inconsistency was detected.
3603	0E13	Bad mode. OpenRsFile failed because the mode parameter was invalid.
3604	0E14	Not readable by RSAM. OpenRsFile failed because the specified file cannot be read by RSAM.

Decimal Value	Hexadecimal Value	Meaning
3605	0E15	Wrong mode. The mode, which was specified when the file was opened, does not allow the operation to succeed. For example, mode read does not allow WriteRsRecord to succeed.
3606	0E16	Record too large. The record is too large to fit into the buffer supplied by ReadRsRecord.
3607	0E17	Good record not found. ScanToGoodRsRecord was unable to locate a well-formed record.

Forms

(See form 1148715 for additional information.)

Decimal Value	Hexadecimal Value	Meaning
3700	0E74	Name not found. The form name supplied to OpenForm was not found within the file. Check that the file name and form name are correct for the form you want.
3701	0E75	Bad object file. The file supplied to OpenForm does not appear to be a valid object module. Possibly the file is empty. Check that the file name is correct for the form you want.
3702	0E76	Form too big. The work area supplied to OpenForm was too small to contain the named form. Use FReport to learn the size of the required work area, and make sure you have allocated sufficient space.

Decimal Value	Hexadecimal Value	Meaning
3703	0E74	Form out of bounds. The screen coordinates passed to DisplayForm would result in a part of the form lying outside the frame. Use FReport to learn the required height and width, and make sure that the frame number and coordinates within the frame are correct for these values.
3704	0E78	Form not displayed. A Forms run-time operation (DefaultField, DefaultForm, ReadField, SetFieldAttrs, UndisplayForm, UserFillField, or WriteField) was attempted on a form that had not been displayed with DisplayForm. Make sure that the form was displayed before attempting any of these operations.
3705	0E79	No such field. A Forms run-time operation (DefaultField, GetFieldInfo, ReadField, SetFieldAttrs, UserFillField, or WriteField) was attempted for which the field specified by pbFieldName, cbFieldName, and index does not exist. Use FReport to display the names and allowable indexes for all fields, and make sure the field specification you have supplied is correct.
3706	0E7A	Bad type specification. ReadField or WriteField was supplied with a type code that is not defined in your configuration. Examine the source text of FmRgtd.Asm for a list of defined type codes, and make sure that the codes you are supplying are in this list.

Decimal Value	Hexa-decimal Value	Meaning
3707	0E7B	Bad data size. ReadField was attempted in which the cbMax parameter was incorrect for the type of data being returned, for example, a cbMax of three for type "Binary." Make sure that the size and type of your data area agree.
3708	0E7C	Invalid data. ReadField or WriteField was attempted in which the requested data conversion could not be performed, for example, reading an alphabetic string as type "Binary." For WriteField, make sure the type of the data you are displaying is correct. For ReadField, it is probably appropriate to display an error message and have the user reenter the data.

Virtual Code Segment Management

Decimal Value	Hexa-decimal Value	Meaning
7300	1C84	Overlay already in memory.
7301	1C85	Next overlay does not fit: The swap buffer is not large enough to swap in the next overlay. Increase the size of the swap buffer.
7302	1C86	ROD will not fit: The swap buffer is not large enough. Increase the size of the swap buffer.
7303	1C87	Swap failed: Internal system error detected (possibly a file system error involving the run file).

Decimal Value	Hexa-decimal Value	Meaning
7304	1C88	Inconsistent ProcInfo table: An internal data structure has been corrupted (possibly caused by link time errors reported in the runfile load map).

Communications

8002	1F42	Lost clear to send during transmission. This generally indicates a modem problem.
8003	1F43	Lost carrier during reception. This indicates a problem with the modem or transmission facilities, or at the host computer site.

Master/Cluster Workstation Communications

8100	1FA4	Time out. A workstation no longer responds to polling. In the context of 2780/3780, 8100 also means: the host computer failed to respond to a transmission. Possibly indicates a total break in communications.
8101*	1FA5	Invalid state. Run the crash dump analyzer.
8102*	1FA6	Communications hardware failure. Run the communications diagnostic.

Decimal Value	Hexa-decimal Value	Meaning
8103*	1FA7	Unrecoverable protocol failure detected by the master workstation. A cluster workstation has ceased to obey proper protocol procedure and has defied all attempts to recover (including an attempt to refuse communication with the master workstation). This can be caused by a hardware failure (including cabling) or excessive DMA loading.
8104*	1FA8	Bad DMA buffer address. An error in system initialization has caused the DMA buffer of the Agent Service Process to fall outside the low-order 128k bytes of memory or on an odd-byte boundary.
8105	1FA9	Invalid card bit. An error has occurred in the Cluster Line Protocol Handle.
8106	1FAA	Busy bit IO. An error has occurred in the Cluster Line Protocol Handle.
8109*	1FAD	Unrecoverable protocol failure detected by a cluster workstation. The cluster workstation has determined that the master workstation is no longer obeying proper protocol procedures. This can be caused by a hardware failure (including cabling) or excessive Multibus DMA loading.

Decimal Value	Hexa-decimal Value	Meaning
8111	1FAF	An error in the hardware (SIO or cabling) on the cluster line has caused a temporary inability of the cluster workstation to communicate with the master workstation.
8112*	1FB0	Master workstation disconnect. An unrecoverable protocol failure has occurred at the master workstation and it has refused further communications with this workstation. The most likely cause is a duplicate workstation identification somewhere within the cluster (if so, the workstation with the duplicate identification should have simultaneously crashed with this error). Other possible causes are the same as code 8109.
8113	1FB1	Request block error. An improperly formatted request block was repeatedly sent by a workstation.
8115	1FB3	Bootstrap failure. A protocol failure occurred during the bootstrap process.
8116	1FB4	No IDs. The ID search algorithm was unable to find a free ID. In general, this indicates that the system build performed for the Operating System currently running on the master work-station specified too few IDs for the cluster configuration.
8117	1FB5	ID search failure. The ID search algorithm found a free ID but was unable to lock onto it for use. In general, this indicates a serious hardware or software problem.

Decimal Value	Hexa-decimal Value	Meaning

2780/3780 and 3270

Decimal Value	Hexa-decimal Value	Meaning
8205	200D	Host computer not polling.
8207	200F	Invalid 3270 command. A subsystem error in the 3270 emulator.
8208	2010	Unexpected response. A subsystem error in the 3270 emulator.
8209	2011	Buffer too small. The buffer must be large enough to accept the longest EBCDIC transmission from the host computer, which may be larger than the screen size.
8210	2012	Request from unknown workstation. Open3270Emulator must be the first request issued, and the device address in all subsequent requests must match the one returned from Open3270Emulator.
8211	2013	Too many workstations. The number of device addresses already assigned is equal to the maximum set at system build; no further Open3270Emulator requests can be honored.
8212	2014	Reject. Subsystem error in the 3270 emulator.

Decimal Value	Hexa-decimal Value	Meaning
8213	2015	For 2780/3780: protocol failure during reception.
		For 3270: protocol failure after selection. A valid selection was received; however the normal BSC error recovery procedures were unable to successfully receive a data block from the host computer.
8214	2016	For 2780/3780: protocol failure during transmission.
		For 3270: protocol failure after poll. A valid poll sequence was received; however the normal BSC error recovery procedures were unable to successfully transmit a data block to the host computer.
8218	201A	Reverse interrupt received from host computer. Transmission was terminated.
8219	201B	An attempt was made to sign on when already signed on, or sign off when already signed off or not idle.
8220	201C	Invalid request code for RJE system service.
8221	201D	Communications line disconnected.
8222	201E	Cannot create sequenced file specification. The entire range of sequence numbers (0-65535) was tried.
8223	201F	Invalid configuration file format. Use the Configure RJE command to create a properly formatted file.

		Hexa-	
	Decimal	decimal	
	Value	Value	Meaning

Communications Interrupt Handlers

	Decimal Value	Hexadecimal Value	Meaning
	8400	20D0	Invalid line number. The line number specified in SetCommISR or ResetCommISR must be either 0 or 1.
	8401	20D1	Line in use. The line specified in SetCommISR is being used by the Operating System.

X.25 Packet Access Method

	Decimal Value	Hexadecimal Value	Meaning
	8500	2134	Link level down. The link level of the X.25 Network Gateway system service is not operational. This situation occurs either at power up before communication with the PDN is established, or during operation if an irrecoverable link level error occurs. The X.25 Network Gateway system service link level software should reestablish communication as soon as possible. If the link level remains down for an extended period, an irrecoverable error at the physical level or the link level exists, and a PDN representative should be contacted.
	8501	2135	Packet level down. The packet level of the X.25 Network Gateway system service is not operational. This situation occurs (1) at power up, (2) during operation following a link level failure and subsequent reestablishment of link level communications, or (3) following an irrecoverable packet level error condition. The X.25 Network Gateway system service should reestablish the packet level as soon as possible. If the packet level remains down for an extended period, a PDN representative should be contacted.

Decimal Value	Hexa-decimal Value	Meaning
8502	2136	Maximum number of this request has been queued. Previously submitted requests of this type must be completed before more can be issued. The maximum number of each request type is

     o   NotifyNextIncomingCall requests: the number of virtual circuits per line.

     o   ReadX25Packet requests: two per virtual circuit.

     o   WriteX25Packet requests: five per virtual circuit.

     o   all other packet access method operation requests: one per virtual circuit.

Generally, since the packet level should complete requests in a short period, the request should be resubmitted. If this condition persists, Query-X25Status should be used to examine the state of the X.25 Network Gateway system service to determine the cause of the delay.

Decimal Value	Hexa-decimal Value	Meaning
8503	2137	X.25 Network Gateway system service is busy. Insufficient memory is available for the X.25 Network Gateway system service to process any more requests at this time. In normal operation, the X.25 Network Gateway system service should complete enough requests to free the memory required for new requests. If this error persists, reinstallation of the X.25 Network Gateway with additional memory should be considered.

Decimal Value	Hexa-decimal Value	Meaning
8504	2138	Process termination. All requests were (or shortly will be) returned and all virtual calls were (or shortly will be) cleared because the user's process has terminated.
8505	2139	Bad port parameter. A NotifyNextIncomingCall operation contains a port range with one of two error conditions:

o   The high port number is less than the low port number.

o   The low and/or high port number is not in ASCII digits.

8506	213A	No virtual circuit available. An InitiateX25Call operation was received, but all virtual circuits were either in use or out of order.
8507	213B	User-specified time out. A ReadX25Packet or a Notify-NextIncomingCall operation could not be fulfilled by the packet level during the specified maximum time.
8508	213C	Virtual circuit in use. A request was received for a virtual circuit (or permanent virtual circuit) in use by some other user.
8509	213D	Call collision. An incoming call was received on a virtual circuit before an InitiateX25Call operation that had been allocated· to that virtual circuit could be completed. The process should resubmit the InitiateX25Call operation.

Decimal Value	Hexa-decimal Value	Meaning
8510	213E	Call cleared. An AcceptX25Call operation was made on a circuit for which no call was pending.
8511	213F	Virtual circuit not in use. A request was received for a virtual circuit (or permanent virtual cicuit) that was not allocated to any user.
8512	2140	DTE clear. Either an erroneous packet was received from the PDN, or the process requested that the call be terminated. The X.25 Network Gateway system service cleared the call that was on this virtual circuit. Data in the process of being transferred may have been lost.
8513	2141	DCE clear. The PDN cleared the call that was on this virtual circuit. Data in the process of being transferred may have been lost.
8514	2142	DTE reset. Either an erroneous packet was received from the PDN, or the process requested the call be reset. The X.25 Network Gateway system service reset the call on this virtual circuit. Data in the process of being transferred may have been lost.
8515	2143	DCE reset. The PDN reset the call on this virtual circuit. Data in the process of being transferred may have been lost.

Decimal Value	Hexa-decimal Value	Meaning
8516	2144	DTE restart. An erroneous packet was received from the PDN, and the X.25 Network Gateway system service was restarted. All active calls were cleared. Data in the process of being transferred may have been lost.
8517	2145	DCE restart. The PDN restarted the packet level. All active calls were cleared. Data in the process of being transferred may have been lost.
8518	2146	Virtual circuit not in data transfer mode. A read, write, reset, or interrupt request was received for a virtual circuit that was not in the correct state. Either no call was present, or the circuit was in the process of being cleared or reset.
8519	2147	Interrupt data. This indicates normal completion of a read request, but with an interrupt data packet rather than a normal data packet. Interrupt data are returned to the process before any normal packets being held for the process by the packet level.
8520	2148	Virtual circuit out of order. An irrecoverable error occurred on this virtual circuit, and the X.25 Network Gateway system service declared it out of order. All calls on this circuit were cleared. The circuit can be restored only by the PDN.

Decimal Value	Hexa-decimal Value	Meaning
8521	2149	Internal time out. The PDN did not respond to the packet generated by the request in the required time period. The process should resubmit the request.
8522	214A	Invalid virtual circuit number. Either (1) a request was received for a virtual circuit with a vch parameter that is either out of bounds or is 0 (circuit 0 is reserved), or (2) a ConnectX25Permanent operation was received for a nonpermanent virtual circuit.
8523	214B	Data truncated. Data to be returned to the process exceeded the size of sPacketRet as specified by the process. The data were truncated to the size of the buffer.
8524	214C	No buffer. A read or write operation was attempted, with sBuffer equal to 0.
8525	214D	Permanent circuit. ClearX25Call or AcceptX25Call was issued with the vch parameter of a permanent virtual circuit.

| | Hexa- | |
| Decimal | decimal | |
Value	Value	Meaning

CommIOP

Decimal Value	Hexadecimal Value	Meaning
8601	2199	CommIOP time out. The CommIOP failed to update the status cell within a certain time period. Run the CommIOP diagnostic to determine the cause of the error.
8602	219A	Line not configured. The communications line number is not currently configured in the system. Change the system build parameters.
8603	219B	Missing system image for CommIOP. The file [Sys]<Sys>CommIOP>SysImage.Sys was not found.
8604	219C	CommIOP loading error. The CommIOP could not be loaded successfully. Run the CommIOP diagnostic.
8605	219D	Invalid CommIOP data structure. There is an invalid queue entry, an invalid CommIOP number, etc. Take a crash dump and run the CommIOP diagnostic.
8606	219E	CommIOP channel restart. The carrier problem on the CommIOP channel was cleared.

Decimal Value	Hexa- decimal Value	Meaning
8607	219F	CommIOP channel hold. There is a carrier problem on one of the CommIOP channels. Disconnect cluster workstation one at a time to determine which is failing.
8610	21A2	CommIOP command failure. The CommIOP returned erroneous control information to the master workstation.
8615	21A7	Bad master workstation to CommIOP command. The CommIOP did not recognize the command from the master workstation.
8616	21A8	CommIOP bootstrap checksum failure. The CommIOP checksum test failed while loading its code file from the master workstation.
8617	21A9	CommIOP stacker/destacker failure. The Multibus interface hardware (stacker/destacker) on the CommIOP is not functional.
8618	21AA	Bad CommIOP interrupt. The CommIOP received an interrupt from an unknown source.
8621	21AD	CommIOP RAM failure in write/read test.
8622	21AE	CommIOP RAM failure - invalid bit set.
8623	21AF	CommIOP failure - invalid bit cleared.
8624	21B0	CommIOP RAM failure in addressing test.

Decimal Value	Hexa-decimal Value	Meaning
8631	21B7	CommIOP handler time out. The CommIOP did not get proper status information from the master workstation. The most probable cause is a software problem in the master workstation that caused the master workstation Agent Service Process to be permanently suspended.
8632	21B8	Invalid CommIOP check word. The CommIOP has encountered an invalid check word in its queues. There is probably a memory error in the master workstation.
8633	21B9	CommIOP RAM checksum error. The CommIOP's RAM is probably faulty. Run the CommIOP diagnostic.
8634	21BA	Invalid queue entry. The CommIOP has discovered an invalid queue entry in its data queues. This is possibly a software error.
8635	21BB	Invalid CommIOP buffer pointer. The CommIOP received an invalid memory address of a buffer.
8636	21BC	CommIOP carrier problem.
8637	21BD	CommIOP software inconsistency. This is probably a software error. A crash dump should be taken.
8641	21C1	CommIOP timer failure. The timer hardware on the CommIOP failed the initialization tests.
8642	21C2	CommIOP DMA failure. The DMA hardware on the CommIOP failed the initialization tests.

Decimal Value	Hexa-decimal Value	Meaning
8643	21C3	CommIOP SIO static test failure. The communications hardware on the CommIOP failed the static initialization test.
8644	21C4	CommIOP SIO functional test failure. The communications hardware on the CommIOP failed the functional test.
8701	21FD	Cluster workstation time out. The cluster workstation did not respond in the allotted time period.
8702	21FE	Cluster workstation CRC error. An excessive number of CRC errors were encountered from the cluster workstation. Run the communications and the CommIOP diagnostics.
8703	21FF	Cluster workstation overrun error. The cluster sent too much data per buffer. Check the cluster workstation/master workstation system build parameters.
8704-8712	2200-2208	Bad protocol errors. These errors are probably due to (1) a reset or power down on the cluster workstation or (2) a faulty cluster workstation.
8699	21FB	The cluster is too heavily loaded when the GetClusterStatus operation is invoked.

APPENDIX B
STANDARD CHARACTER SET

GENERAL

Table B-1 below describes the 256-entry character set used when the keyboard is in character mode, the standard encoding is in the Keyboard Encoding Table, and the standard font is in the font RAM. Table B-2 below shows the graphical representation of the characters of Table B-1.

CODE Keys

When the keyboard is in character mode, the two CODE keys are special kinds of SHIFT keys. If either or both is depressed when a non-SHIFT key is pressed, the high-order bit of the key is turned on. For example, CODE-A generates 80h + 61h = 0E1h, CODE-space generates 80h + 20h = 0A0h, etc. Note that any of the values 80h...0FFh can be generated from the keyboard in this way.

In addition, some of the character codes in the range 80h to 0DFh have keyboard encodings that do not require the CODE key.

Legend for Table B-1

Uppercase alphabetics are used for the actual label on the key cap (for example, FINISH, SHIFT).

Lowercase alphabetics are used for descriptions of the key cap label (for example, left arrow) or video display character (for example, dagger).

Where a character can be generated only by depressing a combination of SHIFT and/or CODE and another key, the key combination is shown as a hyphenated list of keys (for example, SHIFT-6).

The four empty key posts covered by the double keys left-SHIFT, right-SHIFT, numeric-0, and NEXT are denoted by (SH-L'), (SH-R'), (0'), and (NEXT'), respectively.

The keys on the numeric pad are denoted "num 0", etc. to distinguish them from the corresponding keys on the typewriter pad.

Table B-1. Standard Character Set

Character Code (hexa- decimal)	Video Display Character	Key
00	null	HELP
01	up arrow	up arrow
02	◣ (triangle)	MARK
03	¢	SHIFT-6
04	filled square	FINISH
05	empty square	PREV PAGE
06	1/2	1/2
07	bell	CANCEL
08	backspace	BACKSPACE
09	tab	TAB
0A	new line	RETURN NEXT
0B	down arrow	down arrow
0C	formfeed	NEXT PAGE
0D	◢ (triangle)	BOUND
0E	left arrow	left arrow
0F	double dagger	MOVE
10	1/4	SHIFT-1/2
11	†	SCROLL UP
12	right arrow	right arrow
13	trough	SCROLL DOWN
14	raised dot	COPY
15	÷	f1
16		(vertical bar)	f2
17	§	f3
18	≠	f4
19	similarity	f5
1A	¶	f6
1B	filled circle	GO
1C	not	f7
1D	≤	f8
1E	∓	f9
1F	≥	f10
20	space	space
21	!	SHIFT-1
22	"	SHIFT-'
23	#	SHIFT-3
24	$	SHIFT-4
25	%	SHIFT-5

Table B-1. Standard Character Set (Cont.)

Character Code (hexa-decimal)	Video Display Character	Key
26	&	SHIFT-7
27	' (single quote)	'
28	(	SHIFT-9
29	)	SHIFT-0
2A	*	SHIFT-8
2B	+	SHIFT-=
2C	, (comma)	,
2D	- (hyphen)	-
2E	. (period)	.
2F	/	/
30	0	0
31	1	1
32	2	2
33	3	3
34	4	4
35	5	5
36	6	6
37	7	7
38	8	8
39	9	9
3A	:	SHIFT-;
3B	;	;
3C	<	SHIFT-[
3D	=	=
3E	>	SHIFT-]
3F	?	SHIFT-/
40	@	SHIFT-2
41	A	SHIFT-a
42	B	SHIFT-b
43	C	SHIFT-c
44	D	SHIFT-d
45	E	SHIFT-e
46	F	SHIFT-f
47	G	SHIFT-g
48	H	SHIFT-h
49	I	SHIFT-i
4A	J	SHIFT-j
4B	K	SHIFT-k
4C	L	SHIFT-l
4D	M	SHIFT-m

Table B-1. Standard Character Set (Cont.)

Character Code (hexadecimal)	Video Display Character	Key
4E	N	SHIFT-n
4F	O	SHIFT-o
50	P	SHIFT-p
51	Q	SHIFT-q
52	R	SHIFT-r
53	S	SHIFT-s
54	T	SHIFT-t
55	U	SHIFT-u
56	V	SHIFT-v
57	W	SHIFT-w
58	X	SHIFT-x
59	Y	SHIFT-y
5A	Z	SHIFT-z
5B	[	[
5C	\ (back slash)	SHIFT-num 8
5D	]	]
5E	∧ (caret)	∧
5F	_ (underline)	SHIFT--
60	` (reverse accent)	SHIFT-num 1
61	a	a
62	b	b
63	c	c
64	d	d
65	e	e
66	f	f
67	g	g
68	h	h
69	i	i
6A	j	j
6B	k	k
6C	l	l
6D	m	m
6E	n	n
6F	o	o
70	p	p
71	q	q
72	r	r
73	s	s
74	t	t

Character Code (hexa-decimal)	Video Display Character	Key
75	u	u
76	v	v
77	w	w
78	x	x
79	y	y
7A	z	z
7B	{	SHIFT-num 4
7C	¦ (broken vertical bar)	SHIFT-num 7
7D	}	SHIFT-num 5
7E	~ (tilde)	SHIFT-⌃
7F	filled rectangle	DELETE
80	null	CODE-HELP (SH-L')
81	I	CODE-up arrow SHIFT (SH-L')
82	I_0	CODE-MARK (SH-R')
83	I_1	CODE-SHIFT-6 SHIFT (SH-R')
84	I_2	CODE-FINISH (O')
85	I_3	CODE-PREV PAGE SHIFT (O')
86	I_4	CODE-1/2 (NEXT')
87	I_5	CODE-CANCEL SHIFT (NEXT')
88	I_6	CODE-BACKSPACE
89	I_7	CODE-TAB
8A	I_8	CODE-RETURN CODE-NEXT
8B	I_9	CODE-down arrow
8C	^{0} (superscript)	CODE-NEXT PAGE
8D	^{1} (superscript)	CODE-BOUND
8E	^{2} (superscript)	CODE-left arrow

Table B-1. Standard Character Set (Cont.)

Character Code (hexa-decimal)	Video Display Character	Key
8F	^{3} (superscript)	CODE-MOVE
90	^{4} (superscript)	CODE-SHIFT-1/2
91	^{5} (superscript)	CODE-SCROLL UP
92	^{6} (superscript)	CODE-right arrow
93	^{7} (superscript)	CODE-SCROLL DOWN
94	^{8} (superscript)	CODE-COPY
95	^{9} (superscript)	CODE-f1
96	$_0$ (subscript)	CODE-f2
97	$_1$ (subscript)	CODE-f3
98	$_2$ (subscript)	CODE-f4
99	$_3$ (subscript)	CODE-f5
9A	$_4$ (subscript)	CODE-f6
9B	$_5$ (subscript)	CODE-GO
9C	$_6$ (subscript)	CODE-f7
9D	$_7$ (subscript)	CODE-f8
9E	$_8$ (subscript)	CODE-f9
9F	$_9$ (subscript)	CODE-f10
A0	A circle	CODE-space
A1	a circle	CODE-SHIFT-1
A2	A umlaut	CODE-SHIFT-'
A3	a umlaut	CODE-SHIFT-3
A4	O umlaut	CODE-SHIFT-4
A5	o umlaut	CODE-SHIFT-5
A6	O slashed	CODE-SHIFT-7
A7	o slashed	CODE-'
A8	U umlaut	CODE-SHIFT-9
A9	u umlaut	CODE-SHIFT-0
AA	c cedilla	CODE-SHIFT-8
AB	e circumflex	CODE-SHIFT-=
AC	e grave	CODE-,
AD	e acute	CODE-—
AE	AE ligature	CODE-.
AF	ae ligature	CODE-/
B0	β	CODE-0
B1	£	CODE-1
B2	degree	CODE-2

Table B-1. Standard Character Set (Cont.)

Character Code (hexa-decimal)	Video Display Character	Key
B3	©	CODE-3
B4	®	CODE-4
B5	™	CODE-5
B6	¹	CODE-6
B7	¹ 1	CODE-7
B8	¹ 2	CODE-8
B9	¹ 3	CODE-9
BA	¹ 4	CODE-SHIFT-;
BB	¹ 5	CODE-;
BC	¹ 6	CODE-SHIFT-[
BD	¹ 7	CODE-=
BE	¹ 8	CODE-SHIFT-]
BF	¹ 9	CODE-SHIFT-/
C0	see Table B-2	CODE-SHIFT-2 SHIFT-HELP
C1	see Table B-2	CODE-SHIFT-a SHIFT-up arrow
C2	see Table B-2	CODE-SHIFT-b SHIFT-MARK
C3	see Table B-2	CODE-SHIFT-c SHIFT-BOUND
C4	see Table B-2	CODE-SHIFT-d SHIFT-FINISH
C5	see Table B-2	CODE-SHIFT-e SHIFT-PREV PAGE
C6	see Table B-2	CODE-SHIFT-f
C7	see Table B-2	CODE-SHIFT-g SHIFT-CANCEL
C8	see Table B-2	CODE-SHIFT-h SHIFT-DELETE
C9	see Table B-2	CODE-SHIFT-i SHIFT-GO
CA	see Table B-2	CODE-SHIFT-j SHIFT-f9
CB	see Table B-2	CODE-SHIFT-k SHIFT-down arrow
CC	see Table B-2	CODE-SHIFT-l SHIFT-NEXT PAGE
CD	see Table B-2	CODE-SHIFT-m

Table B-1. Standard Character Set (Cont.)

Character Code (hexa-decimal)	Video Display Character	Key
CE	see Table B-2	CODE-SHIFT-n SHIFT-left arrow
CF	see Table B-2	CODE-SHIFT-o SHIFT-MOVE
D0	see Table B-2	CODE-SHIFT-p OVERTYPE
D1	see Table B-2	CODE-SHIFT-q SHIFT-SCROLL UP
D2	see Table B-2	CODE-SHIFT-r SHIFT-right arrow
D3	see Table B-2	CODE-SHIFT-s SHIFT-SCROLL DOWN
D4	see Table B-2	CODE-SHIFT-t SHIFT-COPY
D5	see Table B-2	CODE-SHIFT-u SHIFT-f1
D6	see Table B-2	CODE-SHIFT-v SHIFT-f2
D7	see Table B-2	CODE-SHIFT-w SHIFT-f3
D8	see Table B-2	CODE-SHIFT-x SHIFT-f4
D9	see Table B-2	CODE-SHIFT-y SHIFT-f5
DA	see Table B-2	CODE-SHIFT-z SHIFT-f6
DB	see Table B-2	CODE-[
DC	see Table B-2	CODE-SHIFT-num 8 SHIFT-f7
DD	see Table B-2	CODE-] SHIFT-f8
DE	see Table B-2	CODE- ∧
DF	see Table B-2	CODE-SHIFT-- SHIFT-f10
E0	see Table B-2	CODE-SHIFT-num 1
E1	see Table B-2	CODE-a
E2	see Table B-2	CODE-b
E3	see Table B-2	CODE-c
E4	see Table B-2	CODE-d
E5	see Table B-2	CODE-e

Character Code (hexa-decimal)	Video Display Character	Key
E6	see Table B-2	CODE-f
E7	see Table B-2	CODE-g
E8	see Table B-2	CODE-h
E9	see Table B-2	CODE-i
EA	see Table B-2	CODE-j
EB	see Table B-2	CODE-k
EC	see Table B-2	CODE-l
ED	see Table B-2	CODE-m
EE	see Table B-2	CODE-n
EF	see Table B-2	CODE-o
F0	see Table B-2	CODE-p
F1	see Table B-2	CODE-q
F2	see Table B-2	CODE-r
F3	see Table B-2	CODE-s
F4	see Table B-2	CODE-t
F5	see Table B-2	CODE-u
F6	see Table B-2	CODE-v
F7	see Table B-2	CODE-w
F8	see Table B-2	CODE-x
F9	see Table B-2	CODE-y
FA	see Table B-2	CODE-z
FB	see Table B-2	CODE-SHIFT-num 4
FC	see Table B-2	CODE-SHIFT-num 7
FD	bar chart	CODE-SHIFT-num 5
FE	bar chart	CODE-SHIFT-∧
FF	bar chart	CODE-DELETE

Table B-2. Graphic Representation of the Standard Character Set

Character Code (hexadecimal)	Video Display Character	Character Code (hexadecimal)	Video Display Character	Character Code (hexadecimal)	Video Display Character	Character Code (hexadecimal)	Video Display Character	Character Code (hexadecimal)	Video Display Character	Character Code (hexadecimal)	Video Display Character	Character Code (hexadecimal)	Video Display Character	Character Code (hexadecimal)	Video Display Character
00	←	20	(space)	40	@	60	.	80	—	A0	ᾱ	C0	┴	E0	╛
01	◢	21	!	41	A	61	a	81	—₀	A1	ᾱ	C1	—	E1	—
02	⇔	22	"	42	B	62	b	82	—₁	A2	Ħ	C2	+	E2	┴
03	■	23	#	43	C	63	c	83	—₂	A3	̃a	C3	⊥	E3	┼
04	□	24	$	44	D	64	d	84	—₃	A4	Ŏ	C4	⊤	E4	╪
05	⋈	25	%	45	E	65	e	85	—₄	A5	ö	C5	⊤	E5	┱
06	◁	26	&	46	F	66	f	86	—₅	A6	⊘	C6	—	E6	┐
07	⊥	27	'	47	G	67	g	87	—₆	A7	⊘	C7	+	E7	╒
08	⊤	28	(	48	H	68	h	88	—₇	A8	Ü̈	C8	+	E8	▬
09	↘	29	)	49	I	69	i	89	—₈	A9	̈u	C9	⊢	E9	╘
0A	→	2A	*	4A	J	6A	j	8A	—₉	AA	ĉ	CA	┤	EA	⌐
0B	↗	2B	+	4B	K	6B	k	8B	₀	AB	̂e	CB	+	EB	└
0C	◣	2C	,	4C	L	6C	l	8C	₁	AC	̂e	CC	┤	EC	┌
0D	↓	2D	-	4D	M	6D	m	8D	₂	AD	̂e	CD	┤	ED	╤
0E	⊧	2E	.	4E	N	6E	n	8E	₃	AE	⊢	CE	━	EE	╡
0F	⋈	2F	/	4F	O	6F	o	8F	₄	AF	⊢	CF	+	EF	┯
10	⊢	30	0	50	P	70	p	90	₅	B0	⊠	D0	+	F0	╤
11	↑	31	1	51	Q	71	q	91	₆	B1	̊a	D1	⊢	F1	╪
12	⌐	32	2	52	R	72	r	92	₇	B2	⅀	D2	⊣	F2	╫
13	⋅⊦	33	3	53	S	73	s	93	₈	B3	°	D3	⊨	F3	╬
14	—	34	4	54	T	74	t	94	₉	B4	⊙	D4	⊣	F4	╤╪
15	⊘	35	5	55	U	75	u	95	₀	B5	⊗	D5	⊢	F5	‖
16	Ж	36	6	56	V	76	v	96	₁	B6	⌐	D6	⊣	F6	╫
17	≀	37	7	57	W	77	w	97	₂	B7	_	D7	⊣	F7	╬
18	◤	38	8	58	X	78	x	98	₃	B8	ˉ	D8	⊣	F8	│
19	●	39	9	59	Y	79	y	99	₄	B9	ˉ₂	D9	‖	F9	+
1A	⌐	3A	:	5A	Z	7A	z	9A	₅	BA	ˉ₃	DA	‖	FA	┝┥
1B	⋈	3B	;	5B	‸	7B	‹	9B	₆	BB	ˉ₄	DB	┤	FB	⊩
1C	⊣	3C	<	5C	⌐	7C	—	9C	₇	BC	ˉ₅	DC	‖	FC	╫
1D	⇕	3D	=	5D	⌐	7D	‒	9D	₈	BD	ˉ₆	DD	╨	FD	▓
1E	⊹	3E	∧	5E	‹	7E	⌐	9E	₉	BE	ˉ₇	DE	⊢	FE	▓
1F	∧	3F	ʔ	5F	—	7F	■	9F	₀	BF	ˉ₈	DF	⊨	FF	▓

APPENDIX C
KEYBOARD CODES

Table C-1 lists the keyboard codes generated by the keyboard microprocessor. (Refer to legend on page C-2.)

Table C-1. Keyboard Codes Generated by Unencoded Keyboard

Keyboard Code (hexadecimal)	Key	Keyboard Code (hexadecimal)	Key
00	HELP	2A	unused code
01	up arrow	2B	=
02	MARK	2C	, (comma)
03	BOUND	2D	- (hyphen)
04	FINISH	2E	. (period)
05	PREV PAGE	2F	/
06	1/2	30...39	0...9
07	CANCEL	3A	unused code
08	BACKSPACE	3B	;
09	TAB	3C	unused code
0A	RETURN	3D	unused code
0B	down arrow	3E	unused code
0C	NEXT PAGE	3F	invalid code
0D	NEXT	40	indicates the last key released; always has high bit on (that is, 0C0h)
0E	left arrow		
0F	right arrow		
10	(SH-L')	41	num 6
11	SCROLL UP	42	num -
12	MOVE	43	ACTION
13	SCROLL DOWN	44	OVERTYPE
14	COPY	45	LOCK
15	f1	46	num 2
16	f2	47	num 3
17	f3	48	left SHIFT
18	f4	49	right SHIFT
19	f5	4A	num 0
1A	f6	4B	num .
1B	GO	4C	left CODE
1C	f7	4D	right CODE
1D	f8	4E...5A	unused code
1E	f9	5B	[
1F	f10	5C	num 7
20	space	5D	]
21	num 9	5E	∧ (caret)
22	(SH-R')	5F	unused code
23	(0')	60	num 1
24	(NEXT')	61...7A	a...z
25	unused code	7B	num 4
26	unused code	7C	num 8
27	' (single quote)	7D	num 5
28	unused code	7E	unused code
29	unused code	7F	DELETE

Legend for Table C-1

Uppercase alphabetics are used for the actual label on the key cap (for example, FINISH, SHIFT).

Lowercase alphabetics are used for descriptions of the label (for example, left arrow).

The four empty key posts covered by the double keys left-SHIFT, right-SHIFT, numeric-0, and NEXT are denoted by (SH-L'), (SH-R'), (0'), and (NEXT'), respectively.

The keys on the numeric pad are denoted "num 0", etc. to distinguish them from the corresponding keys on the typewriter pad.

APPENDIX D
REQUEST CODES IN NUMERIC SEQUENCE

The request codes listed in this appendix (and up to 7FFFh) are reserved for future expansion. THESE CODES SHOULD NOT BE USED BY SYSTEM BUILDERS. Request codes 8000h through 0FFFFh are available for system-builder use.

Request Code	Operation Name
00	(illegal)
01	SetPath
02	ClearPath
03	SetPrefix
04	OpenFile
05	CreateFile
06	DeleteFile
07	RenameFile
08	GetFileStatus
09	SetFileStatus
10	CloseFile
11	MountVolume
12	DismountVolume
13	ChangeFileLength
14	GetDateTime
15	GetVHB
16	SetDevParams
17	CreateDir
18	DeleteDir
19	CloseAllFiles
20	QuietIO (internal)
21	QueryVidHdw
22	LoadFontRam
23	LoadStyleRam
24	LoadCursorRam
25	ReadDirSector
26	(reserved)
27	GetUCB
28	Chain
29	LoadTask
30	SetFhLongevity
31	GetFhLongevity
32	ResetSubsys (internal)
33	(reserved)
34	(reserved)
35	Read
36	Write
37	DeviceReadIdAndData (internal)
38	Format

Request Code	Operation Name
39	DeviceReadId (internal)
40	AllocExch
41	DeallocExch
42	AllocMemorySL
43	DeallocMemorySL
44	AllocMemoryLL
45	DeallocMemoryLL
46	AllocAllMemorySL
47	ResetMemoryLL
48	QueryMemAvail
49	OpenRTClock
50	CloseRTClock
51	SetDateTime
52	Beep
53	ReadKbd
54	ReadKbdDirect
55	QueryKbdLeds
56	SetKbdLed
57	SetKbdUnencodedMode
58	QueryKbdState
59	SetSysInMode
60	ReadActionCode
61	QueryWSNum
62	CloseAllFilesLL
63	KbdWakeUp (internal)
64	BeeperOff (internal)
65	SetKbdUnencodedModeReal (internal)
66	KbdResetSysin (internal)
67	DisableActionFinish
68	CheckpointSysIn
69	SetIntHandler
70	ResetKbd (internal)
71	ResetSysIn (internal)
72	ResetAgent (internal)
73	ResetComm (internal)
74	ResetVideo
75	InitVidFrame
76	InitCharMap
77	SetScreenVidAttr
78	CloseISAM
79	CreateISAM
80	DeleteISAM
81	DeleteISAMRecord
82	GetISAMRecords
83	LockISAM
84	ModifyISAMRecord
85	OpenISAM
86	ReadISAMRecordByUri
87	ReadNextISAMRecord
88	ReadUniqueISAMRecord
89	RenameISAM
90	SetISAMProtection
91	SetupISAMIterationKey

Request Code	Operation Name
92	SetupISAMIterationPrefix
93	SetupISAMIterationRange
94	StoreISAMRecord
95	UnlockISAM
96	PurgeISAMUser (internal)
97	OpenFileLL
98	ConvertToSys
99	ServeRq
100	GetClusterStatus
101	SetCommISR
102	ResetCommISR
103	KbAttn3270 (internal)
104	ScreenRead3270 (internal)
105	StatusRead3270 (internal)
106	ReadyForCmd3270 (internal)
107	StartEm3270 (internal)
108	StopEm3270 (internal)
109	CancelRq3270 (internal)
110	ReportStatus3270 (internal)
111	SetVerifyCode (internal)
112	(reserved)
113	(reserved)
114	(reserved)
115	(reserved)
116	(reserved)
117	(reserved)
118	(reserved)
119	(reserved)
120	(reserved)
121	SetLpISR
122	DisableCluster
123	GetRunFileHdr (internal)
124	QueryDCB
125	WriteLog
126	SetCommISRRaw (internal)
127	PurgeISAMTransaction
128	EndISAMTransaction
129	GetISAMRecordsHold
130	HoldISAMRecord
131	ReadISAMRecordByUriHold
132	ReadNextISAMRecordHold
133	ReadUniqueISAMRecordHold
134	ReleaseISAMRecord
135	SetupISAMIteration
136	StartISAMTransaction
137	AddQueueEntry
138	RemoveKeyedQueueEntry
139	ReadNextQueueEntry
140	ReadKeyedQueueEntry
141	MarkNextQueueEntry
142	MarkKeyedQueueEntry
143	RemoveMarkedQueueEntry
144	UnmarkQueueEntry
145	ReWriteMarkedQueueEntry

Request Code	Operation Name
146	EstablishQueueServer
147	TerminateQueueServer
148	PurgeQueueServer (internal)
149	SignoffRJE
150	SignonRJE
151	StatusRJE
152	AcceptCommCall
153	CloseAllCommLines
154	CloseCommLine
155	DialComm
156	DisconnectComm
157	FlushCommBuffer
158	GetCommParameters
159	OpenCommLine
160	ReadComm
161	SetCommParameters
162	WriteComm
163	BreakComm
164	(reserved)
165	NotifyNextIncomingCall
166	AcceptX25Call
167	InitiateX25Call
168	ClearX25Call
169	PurgeX25User
170	ReadX25Packet
171	WriteX5Packet
172	WriteX25Interrupt
173	ResetX25Call
174	QueryX25Status
175	ConnectX25Permanent
176	RemovePartition
177	GetPartitionHandle
178	LoadPrimaryTask
179	TerminatePartitionTasks
180	VacatePartition
181	CreatePartition
182	SetPartitionLock
183	SetPartitionExchange
184	GetPartitionExchange
185	GetPartitionStatus
186	SetExitRunFile
187	QueryExitRunFile
188	ConfigureSpooler
189	SpoolerPassword
190	OpenTape
191	ReadTapeRecords
192	WriteTapeRecords
193	TapeOperation
194	CloseTape
195	PurgeTapeUser
196	TapeStatus

197	ResetSplr (internal)
198	ModifyISAMRecordByKey
199	DeleteISAMRecordByKey
200	LogRemote (internal)
201	VacateParCleanup (internal)
202	GetWSUserName
203	SetWSUserName
204	Reserved
205	Reserved
206	Reserved
207	Reserved
208	Reserved
209	Reserved
210	Reserved
211	Reserved
212	Reserved
213	Reserved
214	Reserved
215	ChangeOpenMode
216	GetDirStatus
217	SetDirStatus

APPENDIX E
DATA STRUCTURES

This appendix describes the following:

o   System Common Address Table

o   Application partition and batch data structures including

     Batch Control Block
     Extended Partition Descriptor
     Extended User Control Block
     Partition Configuration Block
     Partition Descriptor

o   System Configuration Block

SYSTEM COMMON ADDRESS TABLE

The System Common Address Table (SCAT) contains
the 4-byte logical memory address of each of a
number of system data structures. A field whose
name begins with "po" contains the logical memory
address of the offset (from DGroup of the System
Image) of the system data structure rather than
the memory address of the system data structure
itself.   The SCAT, shown in Table E-1 below,
begins at memory location 240h.

Table E-1. System Common Address Table (SCAT)

Memory Location	Field	Description
240h	pSysTime	System date/time structure.
242h	saDGroup	Segment base address of the DGroup of the System Image (1-word field; overlaps second half of pSysTime).
244h	pVCB	Video Control Block.
248h	pRgSysError	System Error Status Block.
24Ch	ppPcbRun	Memory address of the Process Control Block of the currently running process.
250h	pASCB	Application System Control Block.
254h	pVersion	Version is a character string whose length is defined by its own first byte.
258h	prgpCISR	Array of entry points (CS:IPs) of the Communications Interrupt Service Routines.
25Ch	prgQDsCISR	Array of 4-byte entries. The second two bytes of each entry are the segment base address to load into the DS Register when the corresponding CISR is activated. The first two bytes are unused.
260h	pDefaultCISR	First instruction of default CISR.
264h	pcLinkBlks	Word containing the count of available link blocks.

Memory Location	Field	Description
268h	pLinkBlkAvail	First link block on a linked list of link blocks available.
26Ch	pcLinkBlkRes	Word containing the count of reserved link blocks. This is the sum of link blocks reserved at system build for the PSend primitive and those dynamically reserved by the Request primitive for use by the Respond primitive.
270h	prgKbdEncode	Keyboard Encoding Table (the default contents are shown in Appendix B).
274h	poRgExch	Offset to exchange zero.
278h	poRgLinkBlk	Offset to the first link block.
27Ch	poRgODcb	Offset to the first entry in the array of offsets to the Device Control Blocks.
280h	poRgOFcb	Offset to the first entry in the array of offsets to the File Control Blocks.
284h	poRgOUcb	Offset to the first entry in the array of offsets to the User Control Blocks.
288h	poRgOVhb	Offset to the first entry in the array of offsets to the Volume Home Blocks.
28Ch	poRgPcb	Offset to the first Process Control Block.

Table E-1. System Common Address Table (SCAT) (Cont.)

Memory Location	Field	Description
290h	poRgPTrb	Offset to the first entry in the array of memory addresses of active timer request blocks.
294h	pSSET	System Service Exchange Table (the Service Exchange Table for request codes 0-7FFFh).
298h	pUSET	User Service Exchange Table (the Service Exchange Table for request codes 8000h-OFFFFh).
29Ch	pRunQ	Queue of ready-to-run Process Control Blocks.
2B4h	pBootBlock	A 16-word array that contains the information passed to the OS by the bootstrap ROM.
2B8h	pBootDevName	Character string containing the name of the device from which the OS was bootstrapped. The first byte of the string contains the byte count of the string.
2BCh	pCrashDumpDevName	Character string containing the name of the device to which the OS crash dump was written. The first byte of the string contains the byte count of the string.
2C0h	pHdcCntlBlk	Hard Disk (Winchester) Control Block.
2C4h	pFloppyCntlBlk	Floppy Disk Control Block.

Memory Location	Field	Description
2C8h	pConfigBlk	System Configuration Block.
2D4	pSysDeviceNum	System device number.
2D8	pSLSC	System Local Service Code Table (that is, the Local Service Code Table for request codes 0-7FFFh).
2DC	pULSC	User Local Service Code Table (that is, the Local Service Code Table for request codes 8000h-0FFFFh).
2E4	pExtCntlReg	A byte that is a copy of the contents of the external control register on B 21-1, B 21-2, and B 21-3 workstation hardware (B 21-1, B 21-2, and B 21-3 workstations only).
2E8	pGrPortVal	The last byte written to the graphics Multibus port.
2EC	pFontRamBuf	The pointer to the 8k buffer in the Operating System for LoadFontRam operation. It is set to 0 if there is no buffer reserved (B22 work-stations only).

APPLICATION PARTITION AND BATCH DATA STRUCTURES

The application partition and batch data structures are located in each application partition. They provide information about the application system executing in an application partition.

The application partition data structures are:

o   Application System Control Block (described in the "Parameter Management" section),

o   Extended Partition Descriptor,

o   Extended User Control Block,

o   Partition Configuration Block, and

o   Partition Descriptor.

The batch data structure is:

o   Batch Control Block.

The Batch Control Block contains the job name, the batch queue name, the file handle and logical file address of the batch job control file, the SysIn and SysOut byte stream work area and buffers, and information on assigned devices. Its format is shown in Table E-2 below.

Table E-2. Batch Control Block

Offset	Field	Size (bytes)
0	SysInBuffer	512
512	SysOutBuffer	512
1024	cbJobName	1
1025	JobName	12
1037	cbQueueName	1
1038	QueueName	50
1088	fSysInBsOpen	1
1089	SysInBswa	50
1139	fSysOutBsOpen	1
1140	SysOutBswa	51
1191	fhLogLL	2
1193	lfaLogLL	4
1197	cAssDev	2
1199	cbLogicalDev1	1
1200	LogicalDev1	12
1212	cbLogicalDev2	1
1213	LogicalDev2	12
1225	cbLogicalDev3	1
1226	LogicalDev3	12
1238	cbPhysicalDev1	1
1239	PhysicalDev1	78
1317	cbPhysicalDev2	1
1318	PhysicalDev2	78
1396	cbPhysicalDev3	1
1397	PhysicalDev3	78
1475	qeh	4
1479	qehStatus	4
1483	bSequence	1
1484	fSpoolSysOut	1
1485	dateTime	4

The Extended Partition Descriptor contains specifications for the current application file and exit run file. Its format is shown in Table E-3 below.

The Extended User Control Block contains information including the offset of the Partition Descriptor and the exit status code. Its use is transparent to the user.

Table E-3. Extended Partition Descriptor

Offset	Field	Size (bytes)
0	cbCurrentRunFileSpec	1
1	CurrentRunFileSpec	78
79	cbExitRunFileSpec	1
80	ExitRunFileSpec	78
158	cbExitRunFilePswd	1
159	ExitRunFilePswd	12
171	ExitRunFilePriority	1

The Partition Configuration Block contains the offsets of the Extended Partition Descriptor, Batch Control Block, and Application System Control Block. Its format is shown in Table E-4 below.

Table E-4. Partition Configuration Block

Offset	Field	Size (bytes)
0	oExtendedParDesc	2
2	oBCB	2
4	oASCB	2

The Partition Descriptor contains the partition
name, the boundaries of the partition and of its
long- and short-lived memory areas, and internal
links to Partition Decriptors in other parti-
tions. Its format is shown in Table E-5 below.

Table E-5. Partition Descriptor

Offset	Field	Size (bytes)
0	oForwardLink	2
2	oBackwardLink	2
4	saLowBound	2
6	saMinLL	2
8	saCurLL	2
10	saCurSL	2
12	saMaxSL	2
14	saHighBound	2
16	cbParitionName	1
17	PartitionName	12
29	fPartitionVacant	1
30	fPartitionLocked	1
31	PartitionExchange	2

The location of the data structures in the
application partition are shown in Figure E-1
below. For more information, see the
"Application Partition Management" section.
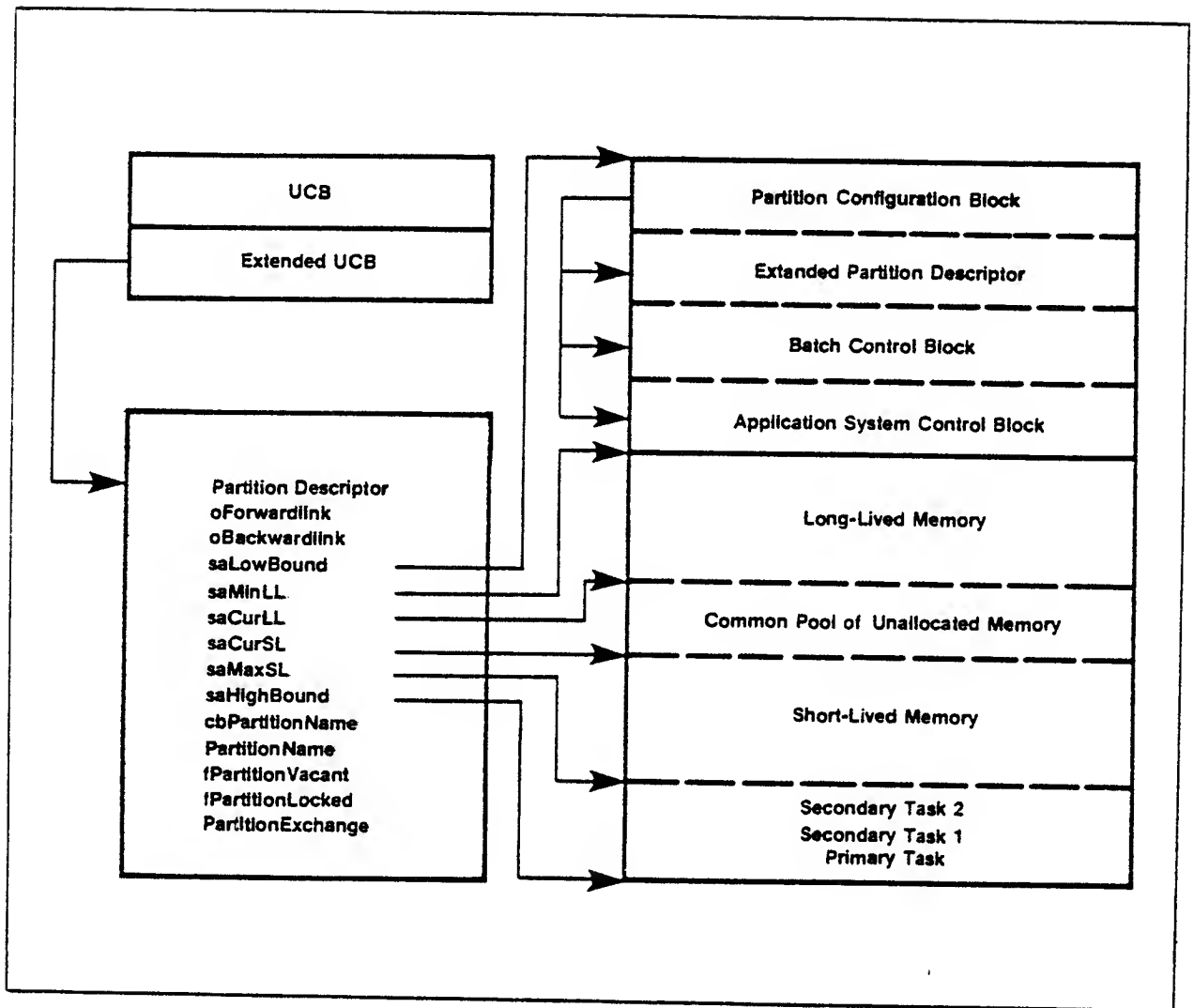
Figure E-1. Application Partition and Batch Data Structure

SYSTEM CONFIGURATION BLOCK

The System Configuration Block contains detailed information about the OS System Image (workstation configuration and system build parameters). The System Configuration Block is located in the system partition. Its address is recorded at address 2C8h in the System Common Address Table.

The format of the System Configuration Block is shown in Table E-6 below.

Table E-6. System Configuration Block

Offset	Field	Size (bytes)	Description
0	System-BuildType	1	Type of system build (used internally).
1	OsType	1	Type of Operating System: 0 swapping; 1 resident.
2	SaMinLL	2	Segment base address of first byte of long-lived memory.
4	SaCurrLL	2	Segment base address of first byte of the common pool of memory.
6	SaCurrSL	2	Segment base address of first byte above common pool of memory.
8	SaMaxSL	2	Segment base address of first byte above short-lived memory.
10	SaMemMax	2	Segment base address of first byte above installed system memory.
12	cPcb	2	Number of Process Control Blocks.
14	cExch	2	Number of exchanges.
16	cLinkBlk	2	Number of link blocks.

Table E-6. System Configuration Block (Cont.)

Offset	Field	Size (bytes)	Description
18	cLinkBlkRes	2	Number of reserved link blocks.
20	cTrb	2	Number of timer request blocks.
22	cIob	2	Number of I/O Blocks.
24	cFcb	2	Number of File Control Blocks.
26	cVhb	2	Number of Volume Home Blocks.
28	cUcb	2	Number of User Control Blocks.
30	cUfb	2	Number of User File Blocks.
32	HardwareType	1	Workstation model: 0 B22 1 B 21-1 2 B 21-2, B 21-3 3 B 21-4
33	Cluster-Configuration	1	Type of configuration: 0 Standalone; 1 Cluster; 2 Master.
34	fNoFileSystem	1	Local file system or not: 0 With file system; 1 Without file system.

Offset	Field	Size (bytes)	Description
35	fCommIop	1	CommIOP or not: 0 Without CommIOP; 1 With CommIOP.
36	fMultiparti-tion	1	Multiple application partitions or not: 0 Single application partition; 1 Multiple application partitions.

APPENDIX F
ACCESSING SYSTEM OPERATIONS FROM ASSEMBLY LANGUAGE

GENERAL

This Appendix describes (1) accessing OS operations from programs written in assembly language and (2) the conventions for argument-passing, register usage, and segments, classes, and groups. Assembly language examples illustrate both OS access and the conventions.

Argument Passing

The Operating System and object module procedures (such as byte streams) deal with data items and structures of many different sizes, ranging from single-byte items, such as Boolean flags, to multibyte structures, such as request blocks and Byte Stream Work Areas.

Three of these are special: 1-byte, 2-byte, and 4-byte data items. Only these are passed as arguments on the stack or returned as results in the registers.

When it is necessary to pass a data structure as an argument, the 4-byte logical memory address of (pointer to) the data structure is used as the argument.

Note that pointers are arranged in memory with the low-order part, the offset, at the lower memory address and the high-order part, the segment base, at the higher memory address. However, the processor architecture of the B 20 System is such that stacks grow from high memory addresses toward low memory addresses. Hence, the high-order part of a pointer is pushed before the low-order part.

Also note that byte arguments are pushed on to the stack as words, with the low-order byte of the word being the argument.

If the argument is Boolean, then the convention is to use a byte value of 0FFh for true and 0 for false. This is not simply nonzero or 0, as the actual test used is to see if the least significant bit is set or clear.

The segments included in the group DGroup are:

Segment	Class
Const	Const
Statics	Const
Data	Data
Stack	Stack
Memory	Memory

Example 1

The TypeSector program copies the first sector of a file to the video display using OS file system operations to open and read the file and SAM (the Sequential Access Method) to write to the video display. The file specification used is obtained from the Executive. The program assumes the file name is specified in a form like:

```
Command       TypeSector
TypeSector
  File name   Sample.File
```

The TypeSector program calls ErrorExit and returns to the Executive if an error is detected.

The program consists of two modules, TypeSector and TypeArg. The modules are assembled and linked as follows:

```
Command             Assemble
Assemble
  Source files   TypeSector.Asm
    .
    .
    .


Command             Assemble
Assemble
  Source files   TypeArg.Asm
    .
    .
    .


Command                Link
Link
  Object modules   TypeSector.Obj TypeArg.Obj
  Run file         TypeSector.Run
    .
    .
    .
```

```
;
; Public and external declarations.
;
PUBLIC Main
EXTRN RgParam:FAR, OpenFile:FAR, Read:FAR, CloseFile:FAR
EXTRN WriteByte:FAR, WriteBsRecord:FAR, bsVid:BYTE
EXTRN TypeArg:FAR
EXTRN Exit:FAR, ErrorExit:FAR
;
; Segment register default initialization.
;
ASSUME CS:NOTHING, DS:NOTHING, ES:NOTHING, SS:NOTHING
;
; Segment declarations.
; All segments used are mentioned in the order they are linked.
;
TypeCode SEGMENT PUBLIC 'Code'
TypeCode ENDS

Const SEGMENT PUBLIC 'Const'
Const ENDS

Statics SEGMENT PUBLIC 'Const'
Statics ENDS

Data SEGMENT PUBLIC 'Data'
Data ENDS

Stack SEGMENT STACK 'Stack'
Stack ENDS
;
; Group the segments together for compatibility with
; object modules.
;
DGroup GROUP Const, Statics, Data, Stack
;
; Stack declaration.
; Declare 100h bytes in this module.  See form 1148681 for
; combining stack segments in different modules.  raStackLim is
; placed so that the stack is the size of the sum of all stack
; declarations.
;
Stack SEGMENT
DB 100h DUP (?)
raStackLim LABEL BYTE
Stack ENDS
```

```
;
; Data declarations.
; All of the variables used in this module are declared here.
;
Data SEGMENT
sDataRet DW ?                               ; This is the variable
                                            ; that the  OS   Read
                                            ; calls and the byte
                                            ; stream WriteBsRecord
                                            ; uses to fill in the
                                            ; actual count of bytes
                                            ; read.

sdRet DB 6 DUP (?)                          ; This is the structure
                                            ; used to obtain
                                            ; parameters from the
                                            ; Executive.  The sdRet
                                            ; structure is defined
                                            ; to be a pointer (four
                                            ; bytes) followed by a
                                            ; count (two bytes).

fh DW ?                                     ; File handle for the
                                            ; source file.


EVEN
rgbBuf DB 512 DUP (?)                       ; Word-aligned input
                                            ; buffer.

DATA ENDS
;
; Macro definition for checking errors.
;
; A procedure of ErcType returns the erc in register AX.  If AX
; is nonzero, then simply call ErrorExit.
;
$SAVE NOGEN
%*DEFINE(CheckErc)LOCAL ok(
     AND          AX, AX
     JE           %ok
     PUSH         AX
     CALL         ErrorExit
%ok:
)
%RESTORE
;
; Main code segment follows.
;
TypeCode SEGMENT
ASSUME CS:TypeCode
Main PROC FAR
```

```
;
; Initialization.
; Set the segment registers (SS, DS) and stack.  The 8086 CPU
; chip disables interrupts for one instruction following a move
; to a segment register, so there is no problem initializing the
; stack pointer (SP).
;
; Since the segment registers SS and DS are being initialized to
; DGroup, DGroup must be explicitly specified when referring to
; the offset of a variable.  If this is not done, then the offset
; of a variable is from the start of the segment in which it is
; declared, not from the start of the group of segments.
;
        MOV             AX, DGroup                  ; Set SS.
        MOV             SS, AX
ASSUME SS:DGroup
        MOV             SP, OFFSET DGroup:raStackLim
        MOV             BP, SP                      ; Set BP for
                                                    ; compatibility with
                                                    ; object
                                                    ; modules.

        PUSH            SS
        POP             DS                          ; Set DS.
ASSUME DS:DGroup
;
; Type the parameter to the video display using TypeArg.
;
; erc := TypeArg (iParam, jParam);
;
        MOV             AX, 1                       ; iParam (1).
        PUSH            AX
        XOR             AX, AX                      ; jParam (0).
        PUSH            AX
        CALL            TypeArg
        %CheckErc
;
; Type a ":" and a new line character.
;
; erc := WriteByte (pBSWA, b);
;
        PUSH            DS                          ; pBSWA (pBsVid).
        MOV             AX, OFFSET DGroup:bsVid
        PUSH            AX
        MOV             AX, ':'                     ; b (:).
        PUSH            AX
        CALL            WriteByte
        %CheckErc
        PUSH            DS                          ; pBSWA (pBsVid).
        MOV             AX, OFFSET DGroup:bsVid
        PUSH            AX
        MOV             AX, 0Ah                     ; b (new line).
        PUSH            AX
```

```
        CALL            WriteByte
    %CheckErc
;
; Get the file name from the Executive.
; (parameter 1, subparameter 0)
;
; erc := RgParam (iParam, jParam, pSdRet);
;
    MOV             AX, 1                       ; iParam (1).
    PUSH            AX
    XOR             AX, AX                      ; jParam (0).
    PUSH            AX
    PUSH            DS                          ; pSdRet.
    MOV             AX, OFFSET DGroup:sdRet
    PUSH            AX
    CALL            RgParam
    %CheckErc
;
; Open the file for mode read.
;
; erc := OpenFile (pFh, pbFileSpec, cbFileSpec, pbPassword,
;                  cbPassword, mode);
;
    PUSH            DS                          ; pFh.
    MOV             AX, OFFSET DGroup:fh
    PUSH            AX
    PUSH            word ptr sdRet + 2          ; pbFileSpec.
    PUSH            word ptr sdRet
    PUSH            word ptr sdRet + 4          ; cbFileSpec.
    XOR             AX, AX                      ; pbPassword (null).
    PUSH            AX
    PUSH            AX
    PUSH            AX                          ; cbPassword (0).
    MOV             AX, 'mr'                    ; mode.
    PUSH            AX
    CALL            OpenFile
    %CheckErc
;
; Read in the first sector (512 bytes).
;
; erc := Read (fh, pBufferRet, sBufferMax, lfa, psDataRet);
;
    PUSH            fh                          ; fh.
    PUSH            DS                          ; pBufferRet.
    MOV             AX, OFFSET DGroup:rgbBuf
    PUSH            AX
    MOV             AX, 512                     ; sBufferMax.
    PUSH            AX
    XOR             AX, AX                      ; lfa (0).
    PUSH            AX
    PUSH            AX
    PUSH            DS                          ; psDataRet.
    MOV             AX, OFFSET DGroup:sDataRet
```

```
        PUSH          AX
        CALL          Read
        %CheckErc
;
; Write the buffer to the video display.
;
; erc := WriteBsRecord (pBSWA, pb, cb, pcbRet);
;
        PUSH          DS                                    ; pBSWA (pBsVid).
        MOV           AX, OFFSET DGroup:bsVid
        PUSH          AX
        PUSH          DS                                    ; pb.
        MOV           AX, OFFSET DGroup:rgbBuf
        PUSH          AX
        MOV           AX, 512                     ; cb.
        PUSH          AX
        PUSH          DS                                    ; pcbRet.
        MOV           AX, OFFSET DGroup:sDataRet
        PUSH          AX
        CALL          WriteBsRecord
        %CheckErc
;
; Return to the Executive.
;
        CALL          Exit
Main ENDP                                         ; End of Main.

TypeCode ENDS                                     ; End of segment.

END Main                                          ; End of module
                                                  ; (specify starting
                                                  ; point as Main).
```

The TypeArg procedure types a parameter passed from the Executive to the video display. It is called with two parameters, iParam and jParam, and is of the type ErcType. It can be called from Pascal as follows:

    erc := TypeArg (iParam, jParam);

The procedure returns 0 if no errors were encountered; otherwise it returns the error in register AX.

The procedure is reentrant and uses no static variables.

This is not a main program but a procedure. It is assumed that the segment registers are properly set before calling this procedure, with DS and SS set to DGroup.

```
;
; Public and external declarations.
;
PUBLIC TypeArg
EXTRN RgParam:FAR, WriteBsRecord:FAR, bsVid:BYTE
;
; The procedure uses 18 bytes of stack for itself, not counting
; calls to other procedures, as follows:
;
;    four bytes for parameters passed to it,
;    four bytes for the return address of the calling service,
;    two bytes to store the BP of the calling service, and
;    eight bytes of local variables.
;
Stack SEGMENT STACK 'Stack'
DB 18 DUP (?)
Stack ENDS
DGroup GROUP Stack

TypeArgCode SEGMENT
ASSUME CS:TypeArgCode, DS:DGroup, ES:NOTHING, SS:DGroup
TypeArg PROC FAR
;
; Set the local variables and parameters as EQUs.
;
```

sArgFrame EQU 4		; Parameters to argument ; are two words (four ; bytes).
sLocalFrame EQU 8		; Eight bytes of local ; variables.
SUB	SP, sLocalFrame	
PUSH	BP	; Save the calling ; service's BP.
MOV	BP, SP	; Use BP as a frame ; pointer.
iParam EQU WORD PTR BP + 16		; First parameter on ; stack.
jParam EQU WORD PTR BP + 14		; Second parameter on ; stack.
sdRet EQU BYTE PTR BP + 2		
pbArg EQU DWORD PTR BP + 2		
cbArg EQU WORD PTR BP + 6		; sdRet is a 6-byte ; structure consisting ; of a pointer (pbArg; ; four bytes) and a ; count (cbArg; two ; bytes) located on the ; stack at SS: BP + 2 ; to SS: BP + 7 .

```
sDataRet EQU WORD PTR   BP + 8                  ; sDataRet is the count
                                                ; of bytes actually
                                                ; written to the video
                                                ; display, and is
                                                ; ignored in this
                                                ; procedure.
;
; erc := RgParam (iParam, jParam, pSdRet);
;
    PUSH        iParam                          ; iParam.
    PUSH        jParam                          ; jParam.
    LEA         AX, sdRet                       ; pSdRet.
    PUSH        SS
    PUSH        AX
    CALL        RgParam
    AND         AX, AX                          ; Check for errors.
    JNZ         Finish
;
; erc := WriteBsRecord (pBSWA, pb, cb, pcbRet);
;
    PUSH        DS                              ; pBSWA (pBsVid).
    MOV         AX, OFFSET DGroup:bsVid
    PUSH        AX
    LES         AX, pbArg                       ; pb.
    PUSH        ES
    PUSH        AX
    PUSH        cbArg                           ; cb.
    LEA         AX, sDataRet                    ; pcbRet.
    PUSH        SS
    PUSH        AX
    CALL        WriteBsRecord
;
; All done, so return erc in AX.
;
Finish:
    POP         BP                              ; Restore the calling
                                                ; service's BP.
    ADD         SP, sLocalFrame                 ; Remove the local
                                                ; variables from the
                                                ; stack.
    RET         sArgFrame                       ; Return with arguments
                                                ; (four bytes) removed
                                                ; from the stack.

TypeArg ENDP
TypeArgCode ENDS
END
```

Example 2

The Timer program uses the Programmable Interval Timer (8253 chip) to generate interrupts and then waits for five interrupts before returning to the Executive.

This example only executes correctly on a standalone workstation because the PIT is used by the OS at cluster and master workstations. This example demonstrates interrupt handling and is not intended for actual use. The SetTimerInt operation is used to control the PIT. (See the "Timer Management" section.)

The Timer program has two parts: the main program and the interrupt handler. The main program sets the interrupt handler as raw using the OS SetIntHandler operation. When this is done, it loops until a flag is set by the interrupt handler, and then displays an * on the video display and resets the flag. When this is done five times, it returns to the Executive.

The interrupt handler, RawTimerHandler, is entered when the 8253 timer counter reaches 0 and generates an interrupt. Because the interrupt is raw, the handler must preserve the register state and send an EOI (end-of-interrupt) to the 8259A when the interrupt service is completed. The handler does this, and also sets the register DS to DGroup before calling the actual service procedure, TimerHandler, which sets a flag and then restarts the timer.

Conversion to Mediated Interrupt Handler

While Timer sets and uses a raw interrupt handler, it can easily be converted to use a mediated interrupt handler. The places in the module that need to be changed are as follows:

1. Change the Equate for fRawInterrupt to FALSE:

    fRawInterrupt EQU FALSE

2. Change the procedure offset pushed on the stack before the call to SetIntHandler to be TimerHandler (instead of RawTimerHandler, which calls TimerHandler):

    ```
    MOV    AX, OFFSET TimerHandler
    PUSH   AX
    ```

3. Delete the handler RawTimerHandler and remove its PUBLIC declaration as it will no longer be used.

Program Assembly and Linking

The program is assembled and linked as follows:

```
Command          Assemble
Assemble
  Source files  Timer.Asm

Command          Link
Link
  Object modules Timer.Obj
  Run file       Timer.Run
    .
    .
    .
```

References

The information required to create the Timer
program is located in the following sections:

OS Interrupt Interface
  "Interrupt Handler" section.

8253 Programmable Interval Timer
  "Interrupt Handler" section.

8259A Interrupt Controller
  "Interrupt Handler" section of this Manual.

WriteByte
  "Sequential Access Method" section.

bsVid
  "Sequential Access Method" section.

```
;  Public and external declarations.  Symbols are made public so
;  that they are included in the symbol file produced by the
;  Linker.
;
PUBLIC, Main, PutChar, RawTimerHandler, TimerHandler
EXTRN WriteByte:FAR, bsVid:BYTE
EXTRN SetIntHandler:FAR
EXTRN Exit:FAR, ErrorExit:FAR
;
;  Segment register default Assume's.
;
ASSUME CS:NOTHING, DS:NOTHING, ES:NOTHING, SS:NOTHING
;
;  Segment declarations.
;  All segments used are mentioned in the order they are linked.
;
TimerCode SEGMENT PUBLIC 'Code'
TimerCode ENDS

Const SEGMENT PUBLIC 'Const'
Const ENDS

Statics SEGMENT PUBLIC 'Const'
Statics ENDS

Data SEGMENT PUBLIC 'Data'
Data ENDS

Stack SEGMENT STACK 'Stack'
Stack ENDS
;
;  Group the segments together for compatibility with object
;  modules.
;
DGroup GROUP Const, Statics, Data, Stack
;
;  Stack declaration.
;  Declare 100h bytes in this module.  See form 1148681 for
;  combining stack segments in different modules.  raStackLim is
;  placed so that the stack is the size of the sum of all stack
;  declarations.  The stack must be large enough for requests to
;  be built by the OS procedural interface, and for the OS to
;  save the process state when the process is swapped out for
;  any reason.
;
Stack SEGMENT
DB 100h DUP (?)
raStackLim LABEL BYTE
Stack ENDS
;
;  Data declarations.
;  All of the variables and constants used in this module are
;  declared here.
```

F-14

```
;
Data SEGMENT
fInterrupt        DW  ?
cInterrupts       DW  ?
cbRet             DW  ?
DATA ends
;
; Equates used in module.
;
; Boolean values.
;
TRUE              EQU  0FFFFh
FALSE             EQU  0
fRawInterrupt     EQU  TRUE
;
; 8253 Timer Equates.
;
iInterrupt        EQU  11
reg8253Cnt0       EQU  28h
reg8253Mode       EQU  2Eh
cmdMode           EQU  30h              ; Select Counter 0, Load
                                        ; lsb, then Msb, Mode 0,
                                        ; Binary Counter 16
                                        ; bits.

bDataLsb          EQU  0FFh
bDataMsb          EQU  0FFh
;
; 8259A Interrupt Equates.
;
reg8259aISR       EQU  20h
reg8259aIMR       EQU  22h
cmdEOI            EQU  20h              ; Nonspecific EOI.
maskTimerOff      EQU  8h               ; This is OR'ed with the
                                        ; value read from IMR
                                        ; (Interrupt Mask
                                        ; Register).
maskTimerOn       EQU  0F7h             ; This is AND'ed with
                                        ; the value read from
                                        ; IMR.
;
; Macro definition for checking errors.
;
; A procedure of ErcType returns the erc in register AX.  If AX
; is nonzero, then simply call ErrorExit.
;
```

```
$SAVE NOGEN
%*DEFINE(CheckErc)LOCAL ok(
    AND         AX, AX
    JE          %ok
    PUSH        AX
    CALL        ErrorExit
%ok:
)
$RESTORE
;
; Main code segment follows.
;
TimerCode SEGMENT
saDGroup DW DGroup
ASSUME CS:TimerCode, DS:NOTHING, SS:NOTHING, ES:NOTHING
Main PROC FAR
;
; Set segment, stack, and frame registers.
;
    MOV         SS, saDGroup
ASSUME SS:DGroup
    MOV         SP, OFFSET DGroup:raStackLim
    MOV         BP, SP
    PUSH        SS
    POP         DS
ASSUME DS:DGroup
;
; Set the interrupt handler.
;
; erc := SetIntHandler(iInt, pIntHandler, saData, fDeviceInt,
;                      fRaw);
;
    MOV         AX, iInterrupt              ; iInt.
    PUSH        AX
    PUSH        CS                          ; pIntHandler.
    MOV         AX, OFFSET RawTimerHandler
    PUSH        AX
    PUSH        DS                          ; saData (not used if
                                            ; fRawInterrupt is
                                            ; TRUE).
    MOV         AX, TRUE                    ; fDeviceInt.
    PUSH        AX
    MOV         AX, fRawInterrupt
    PUSH        AX                          ; fRaw.
    CALL        SetIntHandler
    %CheckErc
;
; Start the 8253 timer.
;
    CLI
    MOV         AL, cmdMode                 ; Initialize counter 0.
    OUT         reg8253Mode, AL
    MOV         AL, bDataLsb
```

```
            OUT             reg8253Cnt0, AL
            MOV             AL, bDataMsb
            OUT             reg8253Cnt0, AL
            IN              AL, reg8259aIMR         ; Turn 8259A mask bit
                                                    ; on.
            AND             AL, maskTimerON
            OUT             reg8259aIMR, AL
            MOV             AX, 5                   ; Initialize counter.
            MOV             cInterrupts, AX
            MOV             AX, FLASE               ; Initialize flag.
            MOV             fInterrupt, AX
            STI
;
; Loop until counter is decremented to 0.
;
MainLoop:
;
; Access to the flag must be with interrupts disabled.
;
            CLI                                     ; Get the flag value.
            MOV             BX, fInterrupt
            MOV             AX, FALSE               ; Reset flag.
            MOV             fInterrupt, AX
            STI

            CMP             BX, TRUE                ; Check for interrupt.
            JNE             MainLoop
;
; Interrupt occurred, so display a character and decrement
; counter.
;
            MOV             AL, '*'
            PUSH            AX
            CALL            PutChar
            DEC             cInterrupts
            JNZ             MainLoop
;
; Received five interrupts, so quit.
;
            CALL            Exit                    ; Return to Executive.
Main ENDP                                           ; End of Main.
;
; PutChar -- Write one character to the video display.
;
; Syntax:
;   MOV             AL, char
;   PUSH            AX
;   CALL            PutChar
;
PutChar PROC NEAR
            PUSH            BP
            MOV             BP, SP
```

```
SArgFrame EQU 2
sRetFrame EQU 2
bChar EQU BYTE PTR SS: BP + sArgFrame + sRetFrame
;
; erc := WriteByte (pBSWA, b);
;
    MOV         AX, SEG bsVid           ; pBSWA.
    PUSH        AX
    MOV         AX, OFFSET bsVid
    PUSH        AX
    MOV         AL, bChar               ; b.
    PUSH        AX
    CALL        WriteByte
    %CheckErc

    POP         BP
    RET         sRetFrame
PutChar ENDP
;
; RawTimerHandler.
; This handler preserves the state of the registers of the
; interrupted process, sets the data segments to DGroup, and
; then calls TimerHandler.  When TimerHandler returns, it
; restores the register state and sends the required EOI to the
; 8259A interrupt controller.
;
; The routine assumes that TimerHandler uses no registers besides
; AX and DX.
;
ASSUME CS:TimerCode, DS:NOTHING, SS:NOTHING
RawTimerHandler PROC FAR
    PUSH        DS                      ; Save DS, AX.
    PUSH        AX

    MOV         DS, saDGroup            ; Put in local DS.
ASSUME DS:DGroup
    CALL        FAR PTR TimerHandler
    MOV         AL, cmdEOI              ; Specify EOI to 8259A.
    OUT         reg8259aISR, AL

    POP         AX                      ; Restore old DS, AX
    POP         DS
ASSUME DS:NOTHING
    IRET                                ; Return from interrupt.
RawTimerHandler ENDP
;
; TimerHandler.
; This procedure sets the flag for main program to look at,
; restarts the timer, and then returns.
;
; The procedure requires DS to be set to DGroup, and it uses AX.
;
```

```
ASSUME CS:TimerCode, DS:DGroup, SS:NOTHING
TimerHandler PROC FAR
    MOV         AX, TRUE                        ; Set interrupt flag.
    MOV         fInterrupt, AX
    MOV         AL, cmdMode                     ; Reinitialize counter.
    OUT         reg8253Mode, AL
    MOV         AL, bDataLsb
    OUT         reg8253Cnt0, AL
    MOV         AL, bDataMsb
    OUT         reg8352Cnt0, AL
    RET
TimerHandler ENDP
TimerCode ENDS                                  ; End of segment.
END Main
```

APPENDIX G
GLOSSARY

$ Directories. The $ Directories are special directories required for the system software to operate correctly. When a request with the directory name of <$> is given as part of a file specification to the OS, the directory name is expanded to the form <$nnn> , where nnn is the user number of the application system.

Action Code. An action code is a key (CANCEL, HELP, 0-9, or f1-f10) depressed in conjunction with the ACTION key. Also see ACTION Key.

ACTION Key. The ACTION key is a special kind of SHIFT key. It is processed specially, even in unencoded mode. The interpretation of all other keys is modified while ACTION is depressed. The key combination ACTION-FINISH terminates the execution of the application system in the primary application partition and invokes the Executive. The key combinations ACTION-A and ACTION-B invoke the Debugger if the Debugger is included in the system at system build. Some of the key combinations that include the ACTION key are available for interpretation by the application system in the primary application partition. This allows the application system to test for special operator intervention without preventing type ahead. Key combinations that include the ACTION key are processed immediately when they are typed. This processing is independent of characters or keyboard codes stored in the type-ahead buffer. Also see Action Code.

Advanced Video Capability. Advanced video capabilities are provided by workstations in the B 22 series with an optional board added to the standard video board. Several versions of this optional board provide various capabilities (for example, bold characters, double-height characters, double-width characters, or a 512 character set) that augment the standard video capabilities of the B 22 series of workstations. Also see Basic Video Capability, Standard Video Capability, and Video Capability.

Agent Service Process. See Cluster Workstation Agent Service Process or Master Workstation Agent Service Process.

Allocation Bit Map.   The Allocation Bit Map controls the assignment of disk sectors. It has 1 bit for every sector on the disk and the bit is set if the sector is available. The Allocation Bit Map is disk-resident.

Application Partition. An application partition is a partition of user memory in which an application system can be executed.   A workstation can have any number of application partitions, with an application system executing concurrently in each.   Also see Primary Application Partition, Secondary Application Partition, and System Partition.

Application Partition Management.   The application partition management facility permits concurrent execution of multiple application systems, each in its own partition. It provides operations for creating, managing, and removing secondary application partitions.   Also see Application Partition, Primary Application Partition, and Secondary Application Partition.

Application Process.   An application process executes code in the application system. It is not a system service process. Also see System Service Process.

Application System. An application system is the collection of all tasks currently in an application partition.   The tasks in an application system access a common set of files and implement a single application. The tasks execute asynchronously.   Also see Task and Application Partition.

Application System Control Block.   The Application System Control Block (ASCB) communicates parameters, the termination code, and other information between an exiting application system and a succeeding application system in the same partition.   Also see the Variable-Length Parameter Block.

Application Workstation.   See B 21 Workstation or B 22 Workstation.
ASCB. See Application System Control Block.

Asynchronous Terminal Emulator. The Asynchronous Terminal Emulator (ATE) utility allows a workstation to emulate an asynchronous character-

oriented ASCII terminal. (See form 1148756 for additional information.)

ATE. See Asynchronous Terminal Emulator.

Bad Sector File. The Bad Sector File contains an entry for each unusable sector of a disk. The Bad Sector File is 1 sector in size.

Banner Page. A banner page is optionally printed by the printer spooler before the printing of each file. The banner page is visually distinctive and also identifies the file being printed. The banner page can contain the text of a notice file. Also see Notice File and Printer Spooler.

Basic Video Capability. Basic video capabilities are provided by the B 21 workstation. Thes capabilities are characterized by an 80-character by 28-line screen, one cursor on the screen, a 256 character set that cannot be modified by software, and a screen split horizontally into multiple frames. Also see Advanced Video Capability, Standard Video Capability, and Video Capability.

Batch Control Block. The Batch Control Block, which is used by the batch manager, contains the job name and class, file handle and logical file address of the batch job control file, Assigned Device Block, and SysIn and SysOut Byte Stream Work Area and buffers. Also see Batch Job Stream and Assigned Device Block.

Batch Job Control File. See Batch Job Stream.

Batch Job Stream. A batch job stream is a file containing batch control statements that is used by the batch manager to direct the execution of noninteractive application systems.

Batch Manager. The batch manager is a system service that uses the batch control statements in a batch job stream to direct the loading and execution of noninteractive application systems.

Batch Partition. A batch partition is an application partition that is under the control of the batch manager. Also see Batch Manager and Batch Job Stream.

Binary Mode. Binary mode is one of three printing mode options in the printer, printer spooler, and communications byte streams. Binary mode does not print the banner page before each file, send extra code not in the file to the printer, nor recognize the escape sequence. Also see Image Mode and Normal Mode.

Blocked. A record file with several records per physical sector is blocked. Also see Record Sequential Access Method and Spanned.

Bootstrap. To bootstrap (or boot) the system is to start it by reloading the Operating System from disk. On other systems, this is often known as Initial Program Load (IPL).

BSWA. See Byte Stream Work Area.

Buffer Management Modes. The Direct Access Method provides two modes of buffer management, write-through and write-behind. Also see Write-Behind Mode and Write-Through Mode.

Byte Stream. A byte stream, a concept of the Sequential Access Method, is a readable (input) or writable (output) sequence of 8-bit bytes. An input byte stream can be read until either the reader chooses to stop reading or it receives status code 1 ("End of File"). An output byte stream can be written until the writer chooses to stop writing. Also see Byte Stream Work Area, Communications Byte Stream, File Byte Stream, Keyboard Byte Stream, Printer Byte Stream, Sequential Access Method, Spooler Byte Stream, Video Byte Stream, and X.25 Byte Stream.

Byte Stream Work Area. The Byte Stream Work Area is a 130-byte memory work area for the exclusive use of Sequential Access Method procedures. Any number of byte streams can be open concurrently, using separate Byte Stream Work Areas. Also see

Byte Stream, Communications Byte Stream, File Byte Stream, Keyboard Byte Stream, Printer Byte Stream, Sequential Access Method, Spooler Byte Stream, Video Byte Stream, and X.25 Byte Stream.

cb. A cb is the count of bytes in a string of bytes.

Character Attribute. A character attribute controls the presentation of a single character. The standard character attributes are reverse video, blinking, half-bright, and underlining. Also see Line Attribute, Screen Attribute, and Video Attributes.

Character Code. In character mode, the 8-bit byte returned by certain keyboard management operations is called a character code (in contrast to the keyboard code returned when the keyboard is in unencoded mode). The character code signifies the depression of a key other than SHIFT, CODE, LOCK, or ACTION. Depression of SHIFT, CODE, and LOCK does not generate a character code, but influences the character codes generated for other keys depressed concurrently. ACTION has a special, system-wide meaning. Also see Character Mode.

Character Map. The character map is the area of memory that holds the coded representation of the characters displayed on the video display. Also see Video Refresh.

Character Mode. In character mode (the default mode), the client process receives an 8-bit character when a key other than SHIFT, CODE, LOCK, or ACTION is pressed. Also see Character Code and Unencoded Mode.

Character Set. See Standard Character Set.

CISR. See Communications Interrupt Service Routine.

Client Process. A client process is a process that makes a request of a system service. Any process, even a OS process, can be a client process since any process can request system services. Also see Queue Manager and System Service Process.

Cluster Configuration. A cluster configuration is a local resource-sharing network consisting of a master workstation and up to 16 cluster workstations. A cluster is connected by one ot four high-speed multidrop half-duplex data links using a variant of the ADCCP/HDLC bit-oriented synchronous protocol. The OS executes in each cluster workstation and in the master workstation. Also the Cluster Workstation, CommIOP, Master Workstation, and Minicluster.

Cluster Workstation. A cluster workstation is a workstation in a cluster configuration and is connected to a master workstation. Also see Cluster and Master Workstation.

Cluster Workstation Agent Service Process. The cluster workstation Agent Service Process converts interprocess requests to interstation messages for transmission to the master workstation. The Agent Service Process is included at system guild in a System Image that is to be used on a cluster workstation. Also see Master Workstation Agent Service Process.

CommIOP. The CommIOP is an intelligent communications processor. The CommIOP serves up to four cluster workstations on each of its two high-speed serial input/output channels. The CommIOP is installed in the Multibus slot of workstations in the B 22 series. CommIOP software consists of: the B 22 series bootstrap-ROM program, the main CommIOP program, and the CommIOP handler.

Code Segment. A code segment is a variable-length (up to 64k bytes) logical entity consisting of reentrant code and containing one or more complete procedures. Also see Data Segment, Segment, and Virtual Code Segment Swapping.

Common Memory Pool. The common memory pool is a single contiguous area of memory in each application partition from which long-lived and short-lived memory segments are allocated.

Communications Byte Stream. A communications byte stream is a byte stream that uses a communications channel. Also see Byte Stream, Byte Stream Work Area, File Byte Stream, Keyboard Byte Stream, Printer Byte Stream, Sequential Access Method, Spooler Byte Stream, Video Byte Stream, and X.25 Byte Stream.

Compact System. A compact system is a version of the Operating System that provides all Operating System functions except for the concurrent execution of multiple application systems. A compact system has a primary application partition and can execute application systems one at a time. An OS is specified to be compact during system build.

Configuration File. A configuration file specifies the characteristics of either the parallel printer, the serial printer, or other device attached to a communications channel. Examples of characteristics are number of characters per line, baud rate, and line control mode (XON/XOFF, CTS). A configuration file is created by the Create Configuration File utility see form 1148772) and is used by printer, printer spooler, and communications byte streams.

Context Switch. A context switch is the saving of register contents when a process is interrupted. When a process is preempted by a process with a higher priority, the OS saves the hardware context of the preempted process in that process's Process Control Block. When the preempted process is rescheduled for execution, the OS restores the contents of the registers. The context switch permits the process to resume as though it were never interrupted. Also see Process, Process Context, and Process Control Block.

Contingency. A contingency can refer to a variety of hardware and software conditions that have undesirable effects. These conditions can be hardware faults such as a memory parity error, inconsistencies detected by the OS such as a bad checksum of a Volume Home Block, or conditions detected by the application system. The OS always terminates execution when it detects an inconsistency.

CPU. The CPU (central processing unit) is the 8086 or 8088 microprocessor.

Crash Dump Area. The Crash Dump Area (the file [Sys]<Sys>CrashDump.Sys) contains a binary memory dump in the event of a system failure.

Cursor RAM. The cursor RAM, part of the advanced video capability, allows software to specify a 10 by 15 bit array as a pattern of pixels in place of the standard cursor (a blinking underline). The cursor bit array is superimposed on the character and blinks.

DAM. See Direct Access Method.

Data Segment. A data segment contains data. It can also contain code, although this is not recommended. If a data segment is shared among processes, concurrency control is the responsibility of those processes. A data segment that is automatically loaded into memory when its containing task image is loaded is called a static data segment to differentiate it from a dynamic data segment that is allocated by a request from the executing process to the memory management facility. Also see Code Segment, Segment, and Task Image.

Date/Time Format. The date/time format provides a compact representation of the date and the time of day that precludes invalid dates and allows simple subtraction to compute the interval between two dates. The date/time format is represented in 32 bits to an accuracy of one second.

DAWA. See Direct Access Work Area.

DCB. See Device Control Block.

Default Response Exchange. Each process is given
a unique default response exchange when it is
created. This special exchange is automatically
used as the response exchange whenever a client
process uses the procedural interface to a system
service. For this reason, the direct use of the
default response exchange is not recommended.
The use of the default response exchange is
limited to requests of a synchronous nature.
That is, the client process, after specifying the
exchange in a Request, must wait for a response
before specifying it again (indirectly or
directly) in another Request. Also see Exchange
and Response Exchange.

Device. A device is a physical hardware
entity. Printers, tape, floppy disks, and
Winchester disks are examples of devices.

Device Control Block. There is a Device Control
Block (DCB) for each physical device. The DCB
contains information, generated at system build,
about the device. For a disk, the information
includes how many tracks are on a disk, the
number of sectors per track, etc. The DCB points
to a chain of I/O Blocks. The DCB is memory-
resident.

Device Password. A device password protects a
device.

Device Specification. A device specification
consists of a devname (device name).

Devname. (Device name) A devname is the only
element of a device specification.

Direct Access Method. The Direct Access Method
provides random access to disk file records
identified by record number. The record size is
specified when the DAM file is created. DAM
supports COBOL Relative I-O, but can also be
called directly from any of the Burroughs
languages. Also see Direct Access Work Area.

Direct Access Work Area. A Direct Access Work
Area is a 64-byte memory work area for the
exclusive use of the Direct Access Method
procedures. Any number of DAM files can be open
simultaneously using separate DAWAs. Also see
Direct Access Method.

Direct Printing. Direct printing transfers text directly from application system partition memory to the specified parallel or serial printer interface of the workstation on which the application system is executing. Direct printing is always accessed through the Sequential Access Method (printer byte streams). Also see Printer Byte Stream, Spooled Printing, and Spooler Byte Stream.

Directory. A directory is a collection of related files on one volume. A directory is protected by a directory password.

Directory Password. A directory password protects a directory on a volume.

Directory Specification. A directory specification consists of a node (node name), volname (volume name), and a dirname (directory name).

Dirname. (Directory name) A dirname is the third element of a directory specification or a full file specification.

Disk Extent. A disk extent is one or more contiguous disk sectors that compose all or part of a file.

DMA. See Direct Memory Access.

Dynamic Data Segment. See Data Segment.

Dynamically Installed System Service. A dynamically installed system service is a system service process that was loaded as an application system and converted itself into a system service using the ConvertToSys operation. (See the "System Services Management" section.) Once installed, a dynamically installed system service has the same capabilities as a system service process that was linked to the System Image. A dynamically installed system service must use OS operations (rather than system build parameters) to identify the request codes that it serves, specify its execution priority, establish its interrupt handlers, etc.

Erc. An Erc is a status (error) code.

Escape Sequence. An escape sequence is a special sequence of characters that invokes special

functions.   Also   see   Printer   Spooler   Escape
Sequence, Submit File Escape Sequence, and Video
Byte Stream.

Event.   In the context of timer management, an
event occurs when an interval elapses.   Also see
System Event.

Event-driven Priority Scheduling.   Event-driven
priority scheduling means that processes are
scheduled for execution based on their priorities
and system events, not on a time limit imposed by
the   OS scheduler.   Also see Process and System
Event.

Exchange.   An exchange is the path over which
messages are communicated from process to process
(or   from   interrupt   handler   to   process).   An
exchange consists of two first-in, first-out
queues, one of processes waiting for messages,
the other of messages for which no process has
yet waited.   An exchange is referred to by a
unique 16-bit integer.   Also see Default Response
Exchange and Response Exchange.

Executive.   An   Executive   is   an   interactive
application program that can be executed in the
primary   application   partition.   It   accepts
commands   from   the   workstation   operator   and
requests   the OS to load tasks to execute those
commands.   This function can be performed by the
Burroughs   Executive   or   by   a   user-written
Executive.   The   Executive   is   loaded   from   the
file   [Sys]<Sys>Exec.Run   if   specified   as   the
SignOnExitFile.   The   file   [Sys]<Sys>Exec.Run
usually   contains   the   Burroughs   Executive;
however,   it   can   contain   a   user-written
Executive.

Exit Run File.   An exit run file is a user-
specified file that is loaded and activated when
an application system exits.   Each application
partition has its own exit run file.

Extended   Partition   Descriptor.   An   Extended
Partition   Descriptor   is   located   in   each
application partition and contains specifications
for the current application file and exit run
file.

Extended User Control Block.   An Extended User
Control   Block   is   located   in   each   applicaton

partititon and contains the offset of the Partition Descriptor. Also see Partition Descriptor.

Extension File Header Blocks. An Extension File Header Block is required for each file that contains more than 32 Disk Extents. Also see File Header Block.

External Interrupt. An external interrupt is caused by conditions that are external to the 8086 processor and are asynchronous to the execution of processor instructions. There are two kinds of external interrupts: maskable and nonmaskable. Also see Internal Interrupt, Maskable Interrupt, and Nonmaskable Interrupt.

FAB. See File Area Block.

FALSE. FALSE is represented in a flag variable as 0.

FCB. See File Control Block.

FHB. See File Header Block.

FIFO. First in, first out.

File. A file is a set of related records (on disk) treated as a unit.

File Access Methods. Several file access methods augment the capabilities of the file management system. The file access methods are object module procedures that are located in the standard OS library and linked to application systems as required. They provide buffering and use the asynchronous input/ouput capabilities of the file management system to automatically overlap input/output and computation. Also see Direct Access Method, Record Sequential Access Method, and Sequential Access Method.

File Area Block. There is a File Area Block for each Disk Extent in an open file. The FAB specifies where the sectors are and how many there are in the Disk Extent. The FAB is pointed to by a File Control Block or another FAB. The FAB is memory-resident. Also see Disk Extent.

File Byte Stream. A file byte stream is a byte stream that uses a file on disk. Also see Byte Stream, Byte Stream Work Area, Communications

Byte Stream, Keyboard Byte Stream, Printer Byte Stream, Sequential Access Method, Spooler Byte Stream, Video Byte Stream, and X.25 Byte Stream.

File Control Block.   There is a File Control Block (FCB) for each open file. The FCB contains information about the file such as the device on which it is located, the user count (that is, how many file handles currently refer to this file), and the file mode (read or modify). The FCB is pointed to by a User Control Block and contains a pointer to a chain of File Area Blocks. The FCB is memory-resident.

File Handle.   A file handle is a 16-bit integer that uniquely identifies an open file. It is returned by the OpenFile operation and is used to refer to the file in subsequent operations such as Read, Write, and DeleteFile.

File Header Block.   There is a File Header Block for each file. The FHB of each file contains information about that file such as its name, password, protection level, the date/time it was created, the date/time it was last modified, the disk address and size of each of its Disk Extents. The FHB is disk-resident and 1 sector in size. Also see Extension File Header Block.

File Password.   A file password protects a file in a directory on a volume.

File Protection Level.   A file protection level specifies the access allowed to a file when the accessing process does not present a valid volume or directory password.

Filename.   (File name) A filename is the fourth element of a full file specification.

Filter Process (User-defined).   A user-defined filter process is a user-written system service process that is included in the System Image at system build. A filter process is interposed between a client process and a system service process that believe they are communicating directly with each other. The Service Exchange Table is adjusted at system build to route requests through the desired filter process. Also see Service Exchange Table.

Filter Process (Local File System).   See Local File System.

Font. A font is a bit array for each of the 256 characters in the character set that defines the representation of each character when displayed on the video display.

Font RAM. The font RAM, part of the video hardware of B22 workstations, contains a 10 by 15 bit array for each of the 256 characters in the character set. The font RAM can be modified under software control. Also see Font ROM.

Font ROM. The font ROM, part of the video hardware of a B21 workstation, contains a 9 by 11 bit array for each of the 256 characters in the character set. Also see Font RAM.

Frame. A frame is a separate, rectangular area of the screen. A frame can have any desired width and height (up to those of the entire screen).

Frame Descriptor. A frame descriptor is a component of the Video Control Block and contains all information about one of the frames. The number of frame descriptors in the Video Control Block is specified at system build. Also see Video Control Block.

Full File Specification. A full file specification consists of a node (node name), volname (volume name), dirname (directory name), and a filename (file name).

Hashing Techniques. See Randomization Techniques.

Image Mode. Image mode is one of three printing options in printer, printer spooler, and communications byte streams. Image mode prints the banner page before each file and recognizes escape sequences but performs no code conversions. Also see Normal Mode and Binary Mode.

Indexed Sequential Access Method. The Indexed Sequential Access Method (ISAM) provides efficient, yet flexible random access to fixed-length records identified by multiple keys stored in disk files. (See form 1148723.)

Input Byte Stream. See Byte Stream.

Internal Interrupt. An internal interrupt (often called a trap) is caused by and is synchronous with the execution of processor instructions. The causes of internal interrupts are an erroneous divide instruction, the 8086 Trap Flag, the INTO (interrupt on overflow) instruction, and the INT (interrupt) instruction. Also see External Interrupt.

Interrupt. An interrupt (external or internal) is an event that interrupts the sequential execution of processor instructions. When an interrupt occurs, the current hardware context (the state of the hardware registers) is saved. This context save is performed partly by the 8086 processor and partly by the Operating System. Also see External Interrupt, Internal Interrupt, Maskable Interrupt, Nonmaskable Interrupt, and Pseudointerrupt.

Interrupt Handler. An interrupt handler is a locus of computation that is given control when an interrupt occurs. Since an interrupt handler is not a process, it is permitted to invoke only a few specific operations. OS interrupt handlers are provided for each interrupt type. Each interrupt handler services all interrupts of a single type. The OS supports two kinds of interrupt handlers, mediated and raw. Also see **Mediated** Interrupt Handler and Raw Interrupt Handler.

Interrupt Levels. On B 22 workstations, the Programmable Interrupt Controller supports eight interrupt levels: 0-Multibus devices, 1-SIO communications controller, 2-Multibus devices, 3-Programmable Interval Timer, 4-printer, keyboard, Real-Time Clock, and high-speed mathematics coprocessor, 5-Multibus devices, 6-Multibus devices, and 7-disk storage subsystem (floppy and Winchester).

Interrupt Type Code. Each potential source of interrupt is assigned an interrupt type code (a number in the range 0 to 119) that is used to vector (direct) the interrupt to the appropriate interrupt handler. Also see Interrupt and Interrupt Handler.

Interrupt Vector Table. The Interrupt Vector Table begins at physical memory address 0 and contains a 4-byte entry for each interrupt type. Each 4-byte entry contains the logical memory address (CS:IP) of the first instruction to be executed when an interrupt of that type occurs. Also see Interrupt Type Code.

IOB. See I/O Block.

I/O Block. The I/O Block (IOB) is used by the OS as temporary storage during Read, Write, and other input/output operations. The IOB contains information obtained from the request block. The number of IOBs specified at system build must be adequate for the maximum number of input/output operations that will be in progress simultaneously. The IOB is memory-resident.

IPC. Interprocess Communication. See the "Interprocess Communication Management" section.

ISAM. See Indexed Sequential Access Method.

ISC. Interstation Communication. See the "Interprocess Communication Management" section.

Kernel. The Kernel is the most primitive and the most powerful component of the OS. It executes with a higher priority than any process but it is not itself a process. The Kernel is responsible for the scheduling of process execution; it also provides interprocess communication primitives.

Keyboard Byte Stream. A keyboard byte stream is a byte stream that uses the keyboard. Also see Byte Stream, Byte Stream Work Area, Communications Byte Stream, File Byte Stream, Printer Byte Stream, Sequential Access Method, Spooler Byte Stream, Submit Facility, Video Byte Stream, and X.25 Byte Stream.

Keyboard Code. In unencoded mode, the 8-bit byte returned by certain keyboard management operations is called a keyboard code. The

keyboard code identifies the key in the low-order seven bits and indicates the direction of key motion in the high-order bit. A 0 indicates key depression; 1 indicates key release. Also see Unencoded Mode and Appendix C of this Manual.

Keyboard Encoding Table. The Keyboard Encoding Table is used in converting the sequence of keyboard codes to 8-bit character codes. The Table controls several aspects of the keyboard code-to-character-code translation: the character code to generate if SHIFT is/is not depressed, whether LOCK has the effect of SHIFT for this key, whether the key is Typematic (repeats), the initial delay before beginning Typematic repeating, and the frequency of Typematic repeating. The Keyboard Encoding Table can be modified dynamically, as well as at system build. See form 1148699 for detailed information on modifying the Keyboard Encoding Table. See Appendix B for the default contents of the Keyboard Encoding Table.

lfa. See Logical File Address.

Line Attribute. A line attribute controls the presentation of a single line. The standard line attribute is cursor position. Also see Character Attribute, Screen Attribute, and Video Attributes.

Link Block. A link block is a system data structure that is used to queue messages at exchanges. Each link block contains the address of the message and the address of the next link block (if any) that is linked onto the exchange. Two pools of link blocks are specified at system build, a general pool, and a special pool used only by the PSend primitive. A call to the Request primitive reserves one link block from the general pool for the corresponding Respond primitive. For these reasons, the number of link blocks in each pool can be specified at system build.

Linker. The Linker utility links one or more object files into a task image stored in a run file. (See form 1148681.)

Local File System. The Local File System allows a cluster workstation to access files on local mass storage as well as files on mass storage at the master workstation. The filter process of

the local file system intercepts each file access request and directs it to the local file system or to the master workstation.

Local Service Code Table. The Local Service Code Table translates each request code to a local service code to specify which of the several services of the system service process is desired. Also see Service Exchange Table.

Log File. The Log File (the file [Sys]<Sys>Log.Sys) is an error-logging file. An entry is placed in the Log File for each recoverable and nonrecoverable device error. This file can be used as a general-purpose logging file, for example, to write entries for accounting information for system services.

Logical File Address. A logical file address is used to locate a particular sector of a file. An lfa specifies the byte position within a file; it is the number (the offset) that would be assigned to a byte in a file if all the bytes were numbered consecutively starting with 0. An lfa is a 32-bit unsigned integer that must be on a sector boundary and is therefore a multiple of 512. For example, the lfa of the third sector of a file is 1024.

Logical Memory Address. (usually abbreviated as memory address) A logical memory address is a 32-bit entity that consists of a 16-bit segment base address and a 16-bit offset. The physical memory address of a byte is computed by multiplying the segment base address by 16 and adding the offset. A byte of memory does not have a unique logical memory address. Rather, any of 4096 combinations of segment base address and offset refer to the same byte of memory.

Long-lived Memory. Long-lived memory is an area of memory in an application partition. It is used for parameters or data passed from an application system to a succeeding application system in the same partition. If a character map other than the one in the system partition is needed, it must be allocated in the long-lived memory area of the primary application

partition. Also see Application Partitions and
System Partitions.

Maskable Interrupt. A maskable interrupt is a
type of external interrupt. A maskable interrupt
is given a priority and controlled by the 8259A
Programmable Interrupt Controller and can be
masked (ignored) by the use of the processor
interrupt-enable flag. A maskable interrupt can
be selectively masked by programming the 8259A
Programmable Interrupt Controller. Also see
External Interrupt and Nonmaskable Interrupt.

Master File Directory. There is an entry for
each directory on the volume in the Master File
Directory (MFD), including the Sys Directory.
The position of an entry within the MFD is
determined by randomization (hashing)
techniques. The entry contains the directory's
name, password, location, and size. The Master
File Directory is disk-resident.

Master Workstation. A master workstation is the
hub of a cluster or minicluster configuration.
The master workstation provides file system,
queue management facility, and other services to
all the cluster workstations. In addition, it
supports its own interactive and batch
application systems. Also see Cluster
Workstation.

Master Workstation Agent Service Process. The
Master Workstation Agent Service Process
reconverts an interstation message to an
interprocess request and queues it at the
exchange of the master workstation system service
process that performs the desired function. Also
see Agent Service Process.

Mediated Interrupt Handler. A mediated interrupt
handler (MIH) is easier to write than a raw
interrupt handler, permits automatic nesting by
priority since processor interrupts are enabled
during its execution, and can communicate its
results to processes through certain Kernel
primitives. Also see Interrupt Handler and Raw
Interrupt Handler.

Memory Address. See Logical Memory Address.

Message. A message is the entity transmitted
between processes by the interprocess
communication facility. It conveys information

and provides synchronization between processes. Although only a single 4-byte data item is literally communicated between processes, this data item is usually the memory address of a larger data structure. The larger data structure is called the message while the 4-byte data item is conventionally called the address of the message. The message can be in any part of memory that is under the control of the sending process. By convention, control of the memory that contains the message is passed along with the message.

MFD. See Master File Directory.

MIH. See Mediated Interrupt Handler.

Minicluster. A minicluster configuration consists of a master workstation and up to four cluster workstations. The master workstation uses its SIO Channel A rather than a CommIOP to connect to the cluster workstations. Also see Cluster, Cluster Workstation, and CommIOP.

Multiprogramming. Multiprogramming is supported at three levels by the OS. First, any number of application systems can coexist, each in its own partition. Second, any number of tasks can be loaded into the memory of the partition and independently executed. Third, any number of processes can independently execute the code of each task. Also see Application System, Process, and Task.

NMI. See Nonmaskable Interrupt.

Node. A node (node name) is the first element of a full file specification. A node is also a standalone or master workstation that is part of a NET Network.

Nonmaskable Interrupt. A nonmaskable interrupt (NMI) is a type of external interrupt. An NMI has a higher priority than a maskable interrupt. An NMI cannot be masked through the use of the processor interrupt-enable flag; however, bits in the Input/Output Control Register allow each of the four conditions that cause NMI to be masked individually. These conditions are write-protect violation, nonexistent or device-addressed memory parity

error, and power failure detection. Also see External Interrupt and Maskable Interrupt.

Nonoverlapped. Nonoverlapped, in the context of file access methods, means that a call to an access method read or write operation does not return to the calling program until all associated input or output is complete.

Normal Mode. Normal Mode is one of three printing options in printer, printer spooler, and communications byte streams. Normal mode prints the banner page before each file, converts tabs into spaces and end-of-line characters to device-dependent codes, and recognizes the escape sequences for manual intervention. Also see Binary Mode and Image Mode.

Notice File. The notice file contains text to be printed on banner pages. The notice file is a convenient way to convey operational information, such as the version of the software currently in use, to a later reader of the printed output. The notice file ([Sys]<Sys>Spooler.Notice) is an ordinary text file that can be created and modified with the Editor or Word Processor. Also see Banner Page.

Object Module Procedure. An object module procedure is a procedure supplied as part of an object module file. It is linked with the user-written object modules of an application system and is not supplied as part of the System Image. Most application systems only require a subset of these procedures. When the application system is linked, the desired procedures are linked together in the run file of the application. The Sequential Access Method is an example of object module procedures. Also see System Common Procedure.

Offset. The offset is the distance, in bytes, of the target location from the beginning of the hardware segment. Also see Logical Memory Address and Physical Memory Address.

Operation. An operation is an OS primitive, service, or procedure.

OS. Operating system.

Output Byte Stream. See Byte Stream.

Overlapped. Overlapped, in the context of file access methods, means that although the application system makes a call to an access method read or write operation and that operation returns, input/output can continue overlapped automatically with the computations of the application system.

Overlay Area. See Swap Buffer.

Paragraph. A paragraph is 16 bytes of memory whose physical memory address is a multiple of 16.

Partition Configuration Block. A Partition Configuration Block is located in each application partition and contains the offsets of the Application System Control Block, Batch Control Block, and Extended Partition Descriptor. Also see Application System Control Block, Batch Control Block, and Extended Partition Descriptor.

Partition Descriptor. A Partition Descriptor is located in each application partition and contains the partition name, the boundaries of the partition and of its long- and short-lived memory areas, and internal links to partition descriptors in other partitions.

Partition Handle. A Partition Handle is a 16-bit integer that uniquely identifies a secondary application partition. It is returned by the CreatePartition operation and is used to refer to the partition in subsequent operations such as LoadPrimaryTask, GetPartitionStatus, and RemovePartition.

pb. A pb is the memory address of a string of bytes.

pb/cb. A pb/cb is a 6-byte entity consisting of the 4-byte memory address of a byte string followed by the 2-byte count of the bytes in that byte string.

PCB. See Process Control Block.

Physical Memory Address. Each byte of memory has a unique 20-bit physical memory address. Software uses logical memory addresses, not physical memory addresses. The physical memory address of a byte is computed by multiplying the

segment base address by 16 and adding the
offset. Also see Logical Memory Address, Offset,
and Segment Base Address.

Physical Record. A physical record (in the
context of file access methods) is an entity that
consists of the record header, the record data,
and the record trailer stored in contiguous
bytes.

PIC. See Programmable Interrupt Controller.

PIT. See Programmable Interval Timer.

Primary Application Partition. The primary
application partition is for interactive programs
that use the keyboard and video display to
interact with the user. Such partitions can be
loaded with interactive programs chosen by the
user, such as the Editor, Word Processor, or
terminal emulators. Also see Secondary
Application Partition.

Primary Task. The primary task is the first task
that is loaded into an application partition. It
is loaded with the LoadPrimaryTask operation by a
process in the primary application partition, or
a Chain, Exit, or ErrorExit operation by a
process in its own partition. The primary task
in turn can load additional tasks, called
secondary tasks, in its own partition with the
LoadTask operation.

Primitive. A primitive is an operation performed
by the Kernel. Also see Kernel.

Printer Byte Stream. A printer byte stream is a
byte stream that performs direct printing. It
can use either a Centronics-compatible printer
connected to a parallel printer port or an
RS-232C-compatible printer connected to
communications Channel A or B of the workstation
on which the application system is executing.
Also see Byte Stream, Byte Stream Work Area,
Communications Byte Stream, Direct Printing, File
Byte Stream, Keyboard Byte Stream, Sequential
Access Method, Spooled Printing, Spooler Byte
Stream, Video Byte Stream; and X.25 Byte Stream.

Printer Spooler. The printer spooler is a
dynamically installed system service that
transfers text from disk files to the printer
interfaces of the workstation on which the

printer spooler is installed. It can simultaneously control the operation of several printers. A disk-based priority-ordered queue controlled by the queue manager contains the file specifications of the files to be printed and the parameters (such as the number of copies and whether to delete the file after printing) controlling the printing. This allows the printer spooler to resume printing automatically when reinstalled following an OS reload. Also see Direct Printing, Printer Byte Stream, Spooled Printing, and Spooler Byte Stream.

Printer Spooler Escape Sequence. Printer spooler escape sequences are special character sequences embedded in text files. They cause the printer to pause when processed by the printer spooler. Escape sequences are available to request a forms change, a print wheel change, and a generic printer pause. The reason for the printer pause (including a text string that is included in the escape sequence) can be ascertained by the Spooler utility. (See the "Printer Spooler Utilities Overview" in form 1148772.) (Also see Escape Sequence.)

Printing Mode. See Binary Mode, Image Mode, and Normal Mode.

Procedural Interface. A procedural interface, is a convenient way to access system services and is compatible with FORTRAN and Pascal, as well as assembly language.

Procedure. A procedure is a subroutine.

Process. A process is the basic entity that competes for access to the processor and which the OS schedules for execution. Associated with a process is the address (CS:IP) of the next instruction to execute on behalf of this process, a copy of the data to be loaded into the processor registers before control is returned to this process, a default response exchange, and a stack. A process is assigned a priority when it is created so that the OS can schedule its execution appropriately.

Process Context. The context of a process is the collection of all information about a process. The context has both hardware and software components. The hardware context of a process consists of values to be loaded into processor

registers when the process is scheduled for execution. This includes the registers that control the location of the process's stack. The software context of a process consists of its default response exchange and the priority at which it is to be scheduled for execution. The combined hardware and software context of a process is maintained in a system data structure called a Process Control Block. Also see Context Switch and Process Control Block.

Process Control Block. The combined hardware and software context of a process is maintained in a system data structure called a Process Control Block. A Process Control Block is the physical representation of a process. Also see Process Context.

Processor. A processor consists of the CPU, memory, and associated circuitry.

Programmable Interrupt Controller. A master Programmable Interrupt Controller is standard on each workstation in the B 22 series and controls eight interrupt levels. Each interrupt level can be connected (wire ORed) to one or more device controllers or to a slave. The use of slave controllers multiples the number of external interrupt sources that can have a unique identity and priority. The PIC is a very flexible hardware entitiy that can operate in a number of modes. The modes established by OS initialization are level (not edge) triggered, fully (not special fully) nested, fixed (not rotating) priority, and not special mask mode. Also see Interrupt and Interrupt Levels.

Programmable Interval Timer. The Programmable Interval Timer provides high-resolution low-overhead activation of user pseudointerrupt handlers. B 21 workstations do not provide a PIT. Also see Real-Time Clock.

Pseudointerrupt. A pseudointerrupt is implemented in software rather than in hardware and in this sense is not really an interrupt. However, a pseudointerrupt causes an interrupt handler to be executed as a real interrupt is and has the same responsibilities and privileges. Also see Interrupt.

Queue Entry.   A queue entry is a formatted request for processing that is added to the specified queue entry file by client processes. A queue entry consists of a number of contiguous 512-byte sectors and has two parts, a control part (40 bytes reserved for the queue manager), and a type-specific part defined by the user. Client and server processes communicate via fields within the queue entry.   Also see Client Process, Queue Entry File, Queue Manager, and Server Process.

Queue Entry File.   A queue entry file contains entries for a single type of processing such as spooled printing, batch processing, or remote job entry.   Each queue entry file represents a priority-ordered, disk-based queue that is controlled by the queue manager.   Also see Queue Manager.

Queue Entry Handle.   A queue entry handle is a 32-bit integer that uniquely identifies a queue entry.   It is returned by the MarkKeyedQueueEntry and MarkNextQueueEntry operations and used in subsequent ReadQueueEntry, RemoveMarked-QueueEntry, RewriteMarkedQueueEntry, and UnmarkQueueEntry operations.

Queue Index File.   The queue index file is a system-wide text file that defines the queues to be used in the system.   The system manager creates the queue index file in the master workstation, entering information such as the name of each queue, its associated queue entry file, the size of its entries, and the type of the queue (for example, printer spooler, RJE, or batch queue).

Queue Manager.   The queue manager controls named, priority-ordered, disk-based queues contained in queue entry files.   It must be installed in the master workstation, either as a system service in the system partition, or in a secondary application partition.   Also see Queue Entry, Queue Entry File, Queue Entry Handle, Queue Index File, Queue Status Block, and System Service.

Queue Status Block   The Queue Status Block is in the control portion of the queue entry that is reserved for the queue manager.   It is referenced by the MarkKeyedQueueEntry, MarkNextQueueEntry, and ReadQueueEntry operations and reports a queue

entry's server user number, priority, and the buffers in which the queue entry handles for the queue entry and the logically following queue entry are stored.

Randomization Techniques. A file entry in a directory (or a directory entry in a Master File Directory) is located by means of the character string that identifies the file (or the directory). The character string is converted to a pseudorandom number which is then converted to the address of the sector where the entry is expected to be located. If the entry is not in the expected sector, then adjacent sectors are searched.

Raw Interrupt Handler. A raw interrupt handler (RIH) provides faster execution than a mediated interrupt handler since the entry in the Interrupt Vector Table points directly to the entry point of the RIH. A RIH is useful for servicing a high-speed non-DMA device that causes an interrupt whenever a byte is to be transferred. Also see Interrupt Handler and Mediated Interrupt Handler.

Ready State. The ready state is one of three states in which a process can exist. A process is in the ready state when it could be running, but a higher priority process is currently running. Any number of processes can be in the ready state at a time. Also see Running State and Waiting State.

Real-Time Clock. The Real-Time Clock (RTC) is used by the OS to provide the current date and time of day and timing of intervals (in units of 100 ms). Also see Programmable Interval Timer.

Record Fragment. A record fragment is a contiguous area of memory within a record. A record fragment is specified using an offset from the beginning of the record and a byte count.

Record Number. A record number specifies the record position relative to the first record in a file. The record number of the first record in a file access method file is 1.

Record Sequential Access Method. The Record Sequential Access Method (RSAM) provides blocked, spanned, overlapped input and output. An RSAM file is a sequence of fixed- or variable-length

records. Files can be opened for read, write, or append operations. Also see Blocked, Record Sequential Work Area, and Spanned.

Record Sequential Work Area. A Record Sequential Work Area is a 150-byte memory work area for the exclusive use of the Record Sequential Access Method procedures. Any number of RSAM files can be open simultaneously using separate RSWAs. Also see Record Sequential Access Method.

Recording File. A recording file, a file used in recording mode, contains a copy of all characters typed at the keyboard while recording mode is active. A recording file can later be used as a submit file to repeat the same sequence of input characters. The use of a recording file and the use of a submit file are mutually exclusive. Also see Recording Mode and Submit File.

Recording Mode. When recording mode is active, all characters typed at the keyboard and read in character mode are written to a recording file, in addition to being returned to the client process. Also see Recording File.

Request. A request requests that an operation be performed by a system service process.

Request Block. A request block is a block of memory provided by the client process that contains highly structured information. (See the "Concepts" section.) The memory address of the request block is provided by the client process during a Request primitive and by the system service process during a Respond primitive. A request block is the "element" that the application system (or the OS) sends to the OS to request that a particular operation be performed.

Request Code. A request code is a unique 16-bit integer that is placed in a request block by a client process. The request code is used by the Request primitive both to route a request to the appropriate system service process and to specify to that process which of the several services it provides is currently requested. Request codes are listed in numeric sequence in Appendix D.

Request Control Block. A Request Control Block is an internal data structure. There is one for each concurrent request. The number of RCBs is a system build parameter.

Response Exchange. A response exchange is the exchange at which the requesting client process waits for the response of a system service. Also see Default Response Exchange and Exchange.

RIH. See Raw Interrupt Handler.

RSAM. See Record Sequential Access Method.

RTC. See Real-Time Clock.

Run File. A run file is created by the Linker and contains a task image. Also see Task Image.

Running State. The runnning state is one of three states in which a process can exist. A process is in the running state when the processor is actually executing its instructions. Only one process can be in the running state at a time. Also see Ready State and Waiting State.

SAM. See Sequential Access Method.

SAMGen. See SAM Generation.

SAM Generation. SAM generation permits the specification of the device-dependent object modules to be linked to an application system. (See form 1148699.)

Screen Attribute. A screen attribute controls the presentation of the entire screen. The standard screen attributes are blank, reverse video (dark characters on a light background), half-bright, number of characters per line (80 or 132), and the presence or absence of character attributes. Also see Character Attribute, Line Attribute, and Video Attributes.

Secondary Application Partition. A secondary application partition is a memory partition that is created and controlled by using operations provided by the application partition management facility. Such partitions are used for noninteractive applications, such as user applications, the batch manager, or system services including the printer spooler, ISAM, and remote job entry. Also see Application Partition, Application Partition Management, and Primary Application Partition.

Secondary Task. A secondary task is a task that is loaded by the primary task. Also see Primary Task.

Security Mode. The security mode causes the printer spooler to pause before printing a file and wait for a password to be entered.

Segment. A segment is a contiguous (usually large) area of memory that consists of an integral number of paragraphs. Segments are usually classified into one of three types: code, static data, or dynamic data. Each kind of segment can be either shared or nonshared. Also see Code Segment and Data Segment.

Segment Base Address. A segment base address is the high-order 16 bits of the 20-bit physical memory address of the first byte of a hardware segment. (The low-order 4 bits are implicitly 0.) The processor segment registers CS, DS, SS, and ES contain segment base addresses. Also see Logical Memory Address and Physical Memory Address.

Sequential Access Method. The Sequential Access Method provides device-independent access to devices (such as the video display, printer, files, and keyboard) by emulating a conceptual, sequential character-oriented device known as a byte stream. Also see Byte Stream, Byte Stream Work Area, Communications Byte Stream, File Byte Stream, Keyboard Byte Stream, Printer Byte Stream, Spooler Byte Stream, Video Byte Stream, and X.25 Byte Stream.

Server Process. A server process (such as the printer spooler, remote job entry, and batch manager) is a system service that has established itself as an active server for a particular queue. Also see Queue Entry and Queue Manager.

Service Exchange. A service exchange is an exchange that is assigned to a system service process at system build. The system service process waits for requests for its services at its service exchange. Also see Service Exchange Table.

Service Exchange Table. The Service Exchange Table is constructed at system build, resides in the System Image, and translates request codes to

service exchanges. Also see Local Service Code Table and Service Exchange.

Service Process. See System Service Process.

Short-lived Memory. Short-lived memory is an area of memory in an application partition. When a task is loaded, the OS allocates short-lived memory to contain its code and data. Short-lived memory can also be allocated directly by a client process in its own partition. Common uses of short-lived memory are input/output buffers and the Pascal heap. Also see Application Partitions.

Size. The size of a data item or structure always refers to the number of bytes contained.

Spanned. A record file in which a record can begin and end in different physical sectors is spanned. Also see Blocked and Record Sequential Access Method.

Spooled Printing. Spooled printing transfers text to a disk file for temporary storage and queues a request through the queue manager for the printer spooler to transfer the text to the first available printer interface under control of the printer spooler. This facilitates sharing of printers by cluster workstations, as well as concurrent interactive computing and printing. Spooled printing can be accessed through the Sequential Access Method (spooler byte streams) and the printer spooler utilities. Also see Direct Printing, Printer Byte Stream, and Spooler Byte Stream.

Spooler Byte Stream. A spooler byte stream automatically creates a uniquely named disk file for temporary text storage. It then transfers the text to the disk file and expands the disk file as necessary. When the spooler byte stream is closed, a request is queued through the queue manager to the printer spooler to print the disk file and delete it after it is printed. This is spooled printing. Also see Byte Stream, Byte Stream Work Area, Communications Byte Stream, Direct Printing, File Byte Stream, Keyboard Byte Stream, Printer Byte Stream, Sequential Access Method, Spooled Printing, Video Byte Stream, and X.25 Byte Stream.

Standard Character Set. The 256-entry standard character set is described in Appendix B. Unless requested otherwise, the OS loads the Keyboard Encoding Table and the font RAM (B 22 workstations) to implement the standard character set. On B 21 workstations, the character set is stored in ROM.

Standard Video Capability. Standard video capabilities are provided by the B 22 series workstations. These capabilities are characterized by a 34-line screen, a software-selectable 80- or 132-character line, one cursor per line, a 256 character set that can be dynamically modified by software, and a screen split horizontally and/or vertically into multiple frames that can overlap each other. Also see Advanced Video Capability, Basic Video Capability, and Video Capability.

Static Data Segment. See Data Segment.

Status Code. A status code reports the success or failure of the requested operation. A status code is stored in a request block by the system service process and is examined by the client process. See Appendix A for a list of status codes.

Style RAM. The style RAM, part of the advanced video capability, contains 16 entries, each of which specifies the presence or absence of each of the video attributes. Entries are selected by the 4-bit values in the character attribute fields of the character map.

Submit Facility. The submit facility permits a sequence of characters from a file to be substituted for characters typed at the keyboard. The use of submit files allows the convenient repetition of command sequences. Also see Submit File.

Submit File. A submit file, a file used in the submit facility, contains the same sequence of characters that would be typed to the desired programs. When a submit file is activated by a request from an application process or a command to the Executive, a character from the file is returned to the application process whenever it requests a character from the keyboard. A recording file and a submit file cannot be used

simultaneously.   Also see Recording File and
Submit Facility.

Submit File Escape Sequence.   A submit file
escape   sequence   consists   of   two   or   three
characters.   The first is the code 03h, which
indicates the presence of an escape sequence.
The second character of the escape sequence is a
code to identify the special function.   The third
character, if present, is an argument to the
function.   Also see Escape Sequence and Submit
File.

Swap Buffer.   The swap buffer is an overlay area
in the memory of an application partition.   The
swap buffer is used to contain all nonresident
code segments.   This buffer must be large enough
to   contain   the   largest   nonresident   code
segment.   A larger buffer permits more code
segments to be kept in the main memory of the
partition and improves system performance.   Also
see Virtual Code Segment Swapping.

SysCmds.   The Executive's command file (SysCmds)
contains information about each command known to
the   Executive.   [Sys]<Sys>SysCmds   is   used   if
there   is   no   SysCmds   file   in   the   Application
System Control Block.   The New Command command is
used to enter additional commands into SysCmds.

Sys.Font.   The [Sys]<Sys>Sys.Font file contains
the font for the standard character set.

System Administrator.   See System Manager.

System Build.   System build is the collective
name for the sequence of actions necessary to
construct   a   customized   OS   System   Image.
System   build   allows   the   specification   of
installation-specific   parameters   and   the
inclusion of user-written system services.   (See
form 1148699.)

System Common Address Table.   The System Common
Address Table contains the 4-byte logical memory
address of each of a number of OS system data
structures.   It starts at physical memory address
240h.   See Appendix E for more information.

System Common Procedure.   A system common
procedure performs a common system function, such
as returning the current date and time.   The code
of the system common procedure is included in the

System Image and is executed in the same context and at the same priority as the invoking process. The Video Access Method, for example, is a system common procedure. Also see Object Module Procedure.

System Configuration Block. The System Configuration Block allows the application system to determine detailed information about the System Image (workstation configuration and system build parameters). See Appendix E for more information.

System Data Structures. System data structures are data areas contained within the OS and necessary for its operation. These structures are often configuration-dependent. A File Control Block and a File Area Block are examples of system data structures.

Sys(tem) Directory. The Sys(tem) Directory of each volume contains entries for system files, including the Bad Sector File, the File Header Blocks, the Master File Directory, the System Image, the Crash Dump Area, the Log File, and the Executive. The Sys Directory is created by the IVolume utility rather than by the CreateDir operation. Also see Sys(tem) Volume.

System Event. A system event affects the executability of a process. Examples of system events are an interrupt from a device controller, Multibus device, timer, or Real-Time Clock, or a message sent from another process. The system event causes a message to be sent to an exchange at which a higher priority process is waiting; this, in turn, causes the OS to reallocate the processor. Also see Event.

System Image. The System Image (the file [Sys]<Sys>SysImage.Sys) contains a run-file copy of the OS.

System Manager. The system manager is the person responsible for planning, generating, extending, and controlling the use of the OS to improve the overall productivity of the installation.

System Memory. System memory is a contiguous area of memory beginning at address 0 that is permanently reserved for use by the OS.

System Partition. A system partition contains the OS or dynamically installed system services. Also see Application Partition.

System Service. A system service is an operation performed by a system service process.

System Service Process. A system service process is an OS process that services and responds to requests from client processes. Both Burroughs - and user-written system service processes can be dynamically installed or linked to the System Image at system build. A system service process is scheduled for execution in the same manner that an application process is scheduled. Also see Application Process and Client Process.

Sys(tem) **Volume.** The OS is bootstrapped from the Sys(tem) Volume. The Sys(tem) Directory of the Sys(tem) Volume contains entries for system files that are not necessary in the Sys Directories of other volumes. These additional entries must be placed in [Sys]<Sys> when the volume is initialized. SysImage.Sys, CrashDump.Sys, and Log.Sys are created (but not initialized) by the IVolume utility. The other file entries are created using the CreateDir operation or the Create Directory command. These system files are the System Images, the Crash Dump Areas, the Log File, the Debugger, the Executive, the Executive's command file, and the standard character font. Also see Crash Dump Area, Executive, Log File, Sys(tem) Directory, and System Image.

Task. A task consists of executable code, data, and one or more processes. The code and data can be unique to the task or shared with other tasks. A task is created by translating source programs into object modules and then linking them together. This results in a task image that is stored on disk in a run file. When requested by a currently active task, such as the Convergent Executive, the OS reads the task image from the run file into the application partition, relocates intersegment references, and schedules it for execution. The new task can coexist with or replace other application tasks. Also see Application System, Run File, and Task Image.

Task Image. A task image is a program stored in a run file that contains code segments and/or

static data segments. Also see Run File and Task.

Text File. A text file is a file in which each byte represents a printable character, or a control character such as tab (09h), new line (0Ah), or formfeed (0Ch).

Time slicing. Time slicing means that processes with the same priority are executed in turn for intervals of 100 ms in round robin fashion. Processes having priorities within a predefined range are subject to time slicing.

Timer Request Block. The Timer Request Block is a data structure shared by the client process and timer management. The TRB defines the interval after which a message is to be sent to a specified exchange. Also see Real-Time Clock.

Trap. See Internal Interrupt.

TRB. See Timer Request Block.

TRUE. TRUE is represented in a flag variable as 0FFh.

Type-Ahead Buffer. The type-ahead buffer stores keyboard characters (or keyboard codes, if in unencoded mode) that have not yet been read by a client process. If the workstation operator types too many characters in advance of processing, the excess characters are discarded. When the client process reads beyond those characters that were buffered successfully, it receives a special status code. The size of the type-ahead buffer is usually 128 characters, but can be changed at system build.

UCB. See User Control Block.

Unencoded Mode. In unencoded mode, the client process receives an indication of each key depression and release. This mode provides maximum flexibility. Also see Character Mode and Keyboard Code.

User Control Block. There is a User Control Block (UCB) for each user number. The UCB contains the default volume, default directory, default password, and default file prefix set by the last SetPath and SetPrefix operations. The UCB is memory-resident.

User File Block. The User File Block contains a pointer to the File Control Block for each open file.

User Number. A user number is a 16-bit integer that uniquely identifies an application system. Each application partition has a different user number. Processes in the same application partition share the same user number. A process obtains its user number with the GetUserNumber operation (see the "Process Management" section.) In the primary application partition of a standalone or master workstation, the user number is always 0.

Utility. A utility is a program designed to perform a common task such as comparing the contents of two files. IVolume, Backup Volume, Restore, Dump, and Maintain File are examples of utilities. (See form 1148772.)

VAM. See Video Access Method.

Variable-Length Parameter Block. The Variable-Length Parameter Block (VLPB) is used by the Executive or batch manager to communicate parameters to a succeeding application system in the partition in which the VLPB is located. The VLPB is created in the long-lived memory of an application partition and its memory address is stored in the Application System Control Block. Also see Application System Control Block.

VCB. See Video Control Block.

VDM. See Video Display Management.

VHB. See Volume Home Block.

Video Access Method. The Video Access Method provides direct access to the characters and attributes of each frame. VAM can put a string of characters anywhere in a frame, specify character attributes for a string of characters, scroll a frame up or down a specified number of lines, position a cursor in a frame, and reset a frame.

Video Attributes. Video attributes control the visual presentation of characters on the screen. There are three kinds of video attributes: screen, line, and character. Also

see Character Attribute, Line Attribute, and
Screen Attribute.

Video Byte Stream. A video byte stream is a byte
stream that uses the video display. Also see
Byte Stream, Byte Stream Work Area,
Communications Byte Stream, File Byte Stream,
Keyboard Byte Stream, Printer Byte Stream,
Sequential Access Method, Spooler Byte Stream,
and X.25 Byte Stream.

Video Capability. The several models of
workstation have varying levels of video
capability: basic, standard, or advanced. Also
see Advanced Video Capability, Basic Video
Capability, and Standard Video Capability.

Video Control Block. The Video Control Block
contains all information known about the video
display, including the location, height, and
width of each frame, and the coordinates at which
the next character is to be stored in the frame
by the Sequential Access Method. The VCB is
located in OS memory at an address recorded in
the System Common Address Table. Also see Frame
Descriptor.

Video Display Management. Video Display
Management provides direct control over the video
hardware. With it, an application system can
determine the level of video capability, load a
new character font into the font RAM, change
screen attributes, stop video refresh, calculate
the amount of memory needed for the character map
based on the desired number of columns and lines
and the presence or absence of character
attributes, initialize each of the frames, and
initialize the character map.

Video Refresh. Video refresh is a hardware
function that reads (using DMA) characters and
line and character attributes from the character
map in memory. It then converts them from the
extended ASCII (8-bit) memory representation to a
bit array by accessing the font RAM (or font
ROM), and displays these bits on the screen as a
pattern of illuminated dots (pixels). Also see
Font RAM and Font ROM.

Virtual Code Segment Swapping. Virtual code
segment swapping is the method of virtual memory
supported by the   OS. The code of each task
is divided into variable-length segments that

reside on disk in a run file.   As the task
executes, only those code segments that are
required at a particular time actually reside in
the main memory of the application partition; the
other code segments remain on disk until they, in
turn, are required.   When a particular code seg-
ment is no longer required, it is simply overlaid
by another code segment.   Also see Code Segment
and Virtual Memory.

Virtual Memory.   Virtual memory is a technique
that makes the apparent size of memory in an
application partition (from the perspective of
the application programmer) greater than its
physical size.   The primary mechanisms for the
implementation of virtual memory are page
swapping and segment swapping.   The OS supports
virtual code segment swapping.   (The use of
program overlays is not considered virtual
memory because it is not transparent to the
application programmer.)   Also see Virtual Code
Segment Swapping.

VLPB.   See Variable-Length Parameter Block.

Volname.   (Volume name)   A volname is the second
element of a full file specification.

Volume.   A volume is the medium of a disk drive
that was formatted and initialized with a volume
name, a password, and volume control structures
such as the Volume Home Block, the File Header
Blocks, the Master File Directory, etc.   A floppy
disk and the medium sealed inside a Winchester
disk are examples of volumes.

Volume Control Structures.   Volume control
structures allow the file management system to
manage (allocate, deallocate, locate, avoid
duplication of) the space on the volume not
already allocated to the volume control
structures themselves.   A volume contains a
number of volume control structures: the Volume
Home Block, the File Header Blocks, the Master
File Directory, and the Allocation Bit Map, among
others.

Volume Home Block. There is a Volume Home Block for each volume. The VHB is the root structure (that is, the starting point for the tree structure) of information on a disk volume. The VHB contains information about the volume such as its name and the date it was created. The VHB also contains pointers to the Log File, the System Image, the Crash Dump Area, the Allocation Bit Map, the Master File Directory, and the File Header Blocks. The VHB is disk-resident and 1 sector in size.

Volume Password. A volume password protects a volume.

Waiting State. The waiting state is one of three states in which a process can exist. A process is in the waiting state when it is waiting at an exchange for a message. A process enters the waiting state when it must synchronize with other processes. A process can only enter the waiting state by voluntarily issuing a Wait Kernel primitive that specifies an exchange at which no messages are currently queued. The process remains in the waiting state until another process (or interrupt handler) issues a Send (or PSend, Request, or Respond) Kernel primitive that specifies the same exchange that was specified by the Wait primitive. Any number of processes can be in the waiting state at a time. Also see Ready State and Running State.

Workstation. A B 22 workstation is a workstation that has standard (or optionally advanced) video capabilities and two (or optionally five) Multibus slots. Also see Advanced Video Capability, and Standard Video Capability.
A B 21 workstation is a station that has basic video capabilities and no multibus slots. Also see Basic Video Capability.

Write-Behind Mode. In write-behind mode, the Direct Access Method writes changed sectors of the buffer to disk only when new sectors are brought into the buffer, the Direct Access Method file is closed, or the mode is changed to write-through. Write-behind mode provides better performance when the Direct Access Method is used to modify records in sequential order. Also see Buffer Management Modes and Write-Through Mode.

Write-Through Mode.   In write-through mode, the
Direct   Access   Method   immediately   writes   the
changed sectors of the buffer to disk whenever a
record is written or deleted.   The Direct Access
Method guarantees that the file content on disk
is   accurate   at   the   completion   of   a   modify
operation.   Also see Buffer Management Modes and
Write-Behind Mode.

X.25 Byte Stream.   An X.25 byte stream is a byte
stream   that   enables   data   transmission   via   the
X.25 network gateway.   Each open X.25 byte stream
corresponds to a virtual circuit that is initiat-
ed when the byte stream is opened and cleared
when the byte stream is closed.   Also see Byte
Stream,   Byte   Stream   Work   Area,   Communications
Byte   Stream,   File   Byte   Stream,   Keyboard   Byte
Stream,   Printer   Byte   Stream,   Sequential   Access
Method, and Video Byte Stream.

INDEX

D

Documentation Evaluation Form

Title: __B 20 Operating System (BTOS) Reference Manual__   Form No: __1162252__

__(Release Level 3.0)__   Date: __August 1983__

Burroughs Corporation is interested in receiving your comments
and suggestions regarding this manual. Comments will be util-
ized in ensuing revisions to improve this manual.

Please check type of Suggestion:

☐ Addition   ☐ Deletion   ☐ Revision   ☐ Error

Comments:

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

From:

Name _____

Title _____

Company _____

Address _____

_____

Phone Number _____   Date _____

Remove form and mail to:

Documentation Dept, - East
Burroughs Corporation
Box CB7
Malvern, PA 19355